

# SAS Programming Efficiency Tips

**Prerequisites:** Basic to intermediate knowledge of SAS, especially data step programming. We will use SAS version 8.0 for Windows in this course, but experience with other versions and platforms should be adequate. You should be able to directly apply the examples presented here to other versions and platforms.

**Description:** This course will consist of a list of ways to improve programming efficiency (with examples of each).

## What is Efficiency?

Minimizing the use of the following resources generally characterizes programming efficiency:

- CPU time (the time your computer takes to perform calculations)
- I/O time (the time your computer takes to read data into memory and write data from the memory to your hard drive)
- Memory
- Data storage
- Programming time

Today's examples will try to cover all of these, with emphasis on the first, second and last resources.

## Measuring Efficiency

By default, SAS produces CPU statistics in the log after running each data or proc step. To get additional performance statistics, use the statement `options fullstimer;`. The statistics this provides vary by platform. In Windows this option provides only two additional statistics, system CPU time and memory.

The `Contents` and `Datasets` procedures provide information about the size of datasets. These tell how many observations a dataset contains, the storage allocated to values of each variable, and give a general idea of how much I/O time will be required to read and write this data.

**Now on to the tips...**

## Minimize I/O Time (A.k.a. read and write only what you need to)

### Tip 1: Avoid Unnecessary Data Steps

#### Example 1:

##### Inefficient

```
data new;
set disk.old;
run;

proc reg;
model x = y z;
run;
```

##### Efficient

```
proc reg data=disk.old;
model x = y z;
run;
```

Some new users do this for fear that somehow their permanent dataset will be overwritten. This won't happen unless you read and write the same dataset, so do not worry. Others do this in order not to have to use the `data=` option in a procedure. Whatever the reason, the dataset `new` is completely unnecessary.

#### Example 2:

##### Inefficient

```
data new;
set disk.old;
if _N_ le 100;
run;

proc print data=new;
run;
```

##### Efficient

```
proc print data=disk.old (obs=100);
run;
```

A new dataset was created for the sole purpose of looking at only the first 100 observations. You can do this using the `obs=` option, without having to create another dataset.

#### Example 3:

##### Inefficient

```
data new;
set old;
where x > 10;
run;

proc means data=new;
var x y z;
run;
```

##### Efficient

```
proc means data=old;
where x > 10;
var x y z;
run;
```

Here, a new dataset was created for the sole purpose of performing a procedure on a subset of data. Instead, use a `where` statement in the procedure to do this. `where` statements can be used in all procedures.

#### Example 4:

##### Inefficient

```
data new;
input x y z;
datalines;
1 2 3
4 5 6
7 8 9
;

data new2;
set new;
w = y**2;
if y < 6 then name='less';
if y ge 6 then name='more';
run;
```

##### Efficient

```
data new;
input x y z;
w = y**2;
if y < 6 then name='less';
if y ge 6 then name='more';
datalines;
1 2 3
4 5 6
7 8 9
;
```

Here, two data steps and two datasets were used to input raw data and then modify it. You can do this all in one data step without creating an additional dataset.

#### Example 5:

##### Inefficient

```
data new;
set old;
where x > 10 & y < 20;
run;

data new2;
set old;
where x < 10 & y > 20;
run;

data new3;
set old;
where x = 10 & y = 20;
run;
```

##### Efficient

```
data new new2 new3;
set old;
if x > 10 & y < 20 then output new;
else if x < 10 & y > 20 then output new2;
else if x = 10 & y = 20 then output new3;
run;
```

Subsetting data from one dataset into multiple datasets can be achieved in one data step instead of many.

## Example 6:

### Inefficient

```
data new;
set disk.old(rename=(w=neww));
label x = "some label";
format y 5.2;
run;
```

### Efficient

```
proc datasets library=disk;
modify old;
rename w=neww;
label x = "some label";
format y 5.2;
run;
```

The `datasets` procedure can perform many housekeeping operations on a dataset, including copying, deleting, and renaming datasets, renaming variables, adding labels or changing formats, and appending datasets to other datasets. It does these operations much more efficiently than using data step programming.

## Tip 2: Read in Only the Variables You Need

### Inefficient

```
data new;
set old;
keep x y z;
(programming statements);
run;
```

### Efficient

```
data new;
set old (keep= x y z);
(programming statements);
run;
```

With this, unnecessary variables are never read into the PDV.

## Tip 3: Store Only the Variables You Need

This is fairly self-explanatory without an example. Use `drop` and `keep` statements in a data step to eliminate unnecessary variables in your dataset, particularly index variables from `do` loops and variables holding values for intermediate calculations. This saves I/O time by not having to read in as many variables when accessing this dataset later and also saves storage space.

## Tip 4: Read in Only the Raw Data You Need

### Inefficient

```
data new;
infile file;
input x 1-2
      y 3-4
      z 5-6;
if z > 10;
run;
```

### Efficient

```
data new;
infile file;
input z 5-6 @;
if z > 10;
input x 1-2 y 3-4;
run;
```

With this, only necessary observations are read into the PDV.

### **Tip 5: Store Data in SAS Datasets**

Instead of storing data in a raw data file and then reading in the raw data every time you want to use it, read in the raw data once and store it as a permanent SAS dataset for later use.

## Execute Only Necessary Statements And In The Correct Order

### Tip 1: Use If-Then/Else Statements Instead of Multiple If Statements

Inefficient

```
data new;
set old;
if 0 le x lt 10 then
  group="kids";
if 10 le x lt 20 then
  group="teens";
if 20 le x lt 30 then
  group="twenties";
run;
```

Efficient

```
data new;
set old;
if 0 le x lt 10 then
  group="kids";
else if x lt 20 then
  group="teens";
else if x lt 30 then
  group="twenties";
run;
```

In the left program, each `if` statement is executed for every observation. In the right program, as soon as an `if` statement is true, subsequent `else/if` statements are not executed, yielding an overall fewer number of statements to execute.

### Tip 2: Arrange If Statements In Decreasing Order of Their Probability of Being True

In the above example, if you knew that most observations fall in the  $20 < x < 30$  group, then put that `if` statement first. This minimizes the number of `if` statements that are read.

### Tip 3: Use a Retain Statement to Initialize Constants

Inefficient

```
data new;
set old;
a = 5;
b = 13;
(programming statements);
run;
```

Efficient

```
data new;
set old;
retain a 5 b 13;
(programming statements);
run;
```

This assigns the values of constants once, instead of every time an observation is read.

## Tip 4: Group Constants Together in Expressions

### Inefficient

```
%let a = 5;
%let b = 6;
%let c = 7;

data new;
set old;
sum = x + &a + y + &b
      + z + &c;
run;
```

### Efficient

```
%let a = 5;
%let b = 6;
%let c = 7;

data new;
set old;
sum = x + y + z + &a + &b + &c;
run;
```

During compilation, SAS evaluates parts of expressions involving only constants. This result is then used during execution. By putting all constants together, SAS evaluates this value once during compilation instead of once every time data step statements are executed for an observation.

## Tip 5: Watch Out For Missing Values

### Example 1:

Say we know that the variable *w* contains many missing values.

### Inefficient

```
data new;
set old;
wyzsum = 26 + w + y + z;
run;
```

### Efficient

```
data new;
set old;
wyzsum = 26 + y + z + w;
run;
```

Every time an operation is performed on a missing value, SAS records the column and line location and how many missing values occurred at that location. If a missing value comes first in an expression, it is propagated through all subsequent operations in the expression, and SAS records this each time. By putting variables likely to contain missing values last, fewer operations are done on missing values, so SAS has to do much less record keeping, and the program runs faster.

### Example 2:

Again, say we know the variable *w* contains many missing values.

### Inefficient

```
data new;
set old;
wyzsum = 26 + y + z + w;
run;
```

### Efficient

```
data new;
set old;
if x > . then wyzsum = 26 + y + z + w;
run;
```

We can improve on the previous example even further, by introducing an `if` statement, so that statements involving a variable with frequent missing values aren't performed unless the value of that variable is non-missing.

### Tip 6: Use SAS Functions Whenever Possible

#### Inefficient

```
data new;
set old;
meanxyz = (x+y+z)/3;
run;
```

#### Efficient

```
data new;
set old;
meanxyz = mean(x, y, z);
run;
```

SAS functions run more efficiently than analogous programming statements, because the code to run them has been optimized, plus these require significantly less typing in some cases. Use them.

### Tip 7: Use SAS Procedures Whenever Possible

Like the above, the code in SAS procedures has been optimized to run most efficiently. Using these will definitely save you tremendous amounts of programming time. Again- use them instead of your own programming statements.

### Tip 8: Put Only Statements Affected by the Loop Index Variable in the Loop

#### Inefficient

```
data new;
set old;
array xyz[3] x y z;
array xyzsq[3] x2 y2 z2;
do i =1 to 3;
xyzsq[i] = xyz[i]**2;
xysum = x + y;
end;
run;
```

#### Efficient

```
data new;
set old;
array xyz[3] x y z;
array xyzsq[3] x2 y2 z2;
do i = 1 to 3;
xyzsq[i] = xyz[i]**2;
end;
xysum = x + y;
run;
```

Statements inside a loop are executed every time the loop is executed. If a statement not including the index variable is included in the loop, it is executed multiple times needlessly.

## Tip 9: Use a Series of `if` Statements Instead of Multiple `and` Operators

### Inefficient

```
data new;
set old;
if x > 5 and y < 6 and
    z ge 20 then var = 'no';
run;
```

### Efficient

```
data new;
set old;
if x > 5 then
    if y < 6 then
        if z ge 20 then var = 'no';
run;
```

With multiple `and` operators, SAS read every condition even if one is false. With multiple `if-then` clauses, SAS stops reading the rest of the statement as soon as one of the clauses is found false.

## Tip 10: Put Loops With the Fewest Iterations Outermost

### Inefficient

```
data new;
set old;
do i = 1 to 100;
do j = 1 to 10;
(programming statements);
end; end;
run;
```

### Efficient

```
data new;
set old;
do j = 1 to 10;
do i = 1 to 100;
(programming statements);
end; end;
run;
```

The total number of iterations of the loops is the same regardless of the order they are in; however, the number of times the iteration variables are incremented is not. In the left program, `j` must be incremented 10 times for every iteration of `i`, and `i` must be incremented 100 times, yielding  $10 \times 100 + 100 = 1,100$  changes of an iteration variable. Likewise, in the right program, `i` must be incremented 100 times for every iteration of `j`, but `j` must be incremented only 10 times, yielding  $100 \times 10 + 10 = 1,010$  changes. The more iterations the outer loop contains, the more times SAS must increment an index variable.

## Tip 11: Use Arrays and Macros for Repeated Statements, But If You Don't, Turn the Macro Facility Off

Using arrays and macros can save you the time of having to repeatedly type groups of statements. Learn how to use these. If your program does not include any macros, however, eliminate the macro facility using the statement `options nomacro;`. This cuts down on CPU time, since SAS doesn't have to keep the macro processor ready to check for ampersands and percent signs that signify macro names.

## Save Storage Space

### Tip 1: Use a Length Statement

For integers and character variables with short values, decreasing the length of these variables may dramatically decrease the size of your dataset. This is achieved with the statement

```
length var_name1 length1 var_name2 length2 etc;
```

Note that this statement must occur in a data step before the first reference to any of the variables included in it.

Non-integer values generally should not be stored in less than 8 bytes, as they will lose precision. The following is a table of lengths for integer values. Notice that the smallest length (for a one-digit number, say) is three bytes.

Length in bytes	Largest Integer Represented Exactly
3	8,192
4	2,097,152
5	536,870,912
6	137,438,953,472
7	35,184,372,088,832
8	9,007,199,254,740,992

For character variables, one character corresponds directly to one byte. Hence, to minimize storage space, set the length of each character variable to the number of characters in the longest value of the variable.

Dates are stored as numeric variables in SAS, and for the years 1700 A.D. – 2200 AD, contain five or less digits. Hence, all date variables can be stored in four bytes.

### Tip 2: Eliminate Temporary Datasets When You Are Done With Them

If you don't need to use a temporary dataset again, you can free disk space by deleting it. Two ways to do this are:

- Reuse the dataset name somewhere later in your program; the new dataset will overwrite the old one.
- Use the `datasets` procedure to delete datasets you no longer need. The syntax to do this is:

```
proc datasets library = name_of_lib;  
delete dataset_name;  
run;
```

### **Tip 3: Consider Storing 0-1 Variables as Character Variables**

The smallest length of a numeric variable containing one-digit values is three bytes; character variables containing one-character values can be stored in one byte. If your data are primarily 0-1 variables, and you don't need to perform numeric calculations on these, consider storing these variables as character variables instead of numeric variables.

### **Tip 4: Shorten Data When Possible**

For repetitive long character values, consider using abbreviated values in the data and then applying the `format` procedure to display longer names when printing. For example, suppose you have the variable `state`. Instead of using "North Carolina" as the data value, use "NC", and apply a format so that the value reads "North Carolina" when printing. This changes the length of this variable from 14 bytes (assuming "North Carolina" is the longest value) to 2 bytes.

## Sort Carefully

### Tip 1: Avoid Unnecessary Sorting

#### Example 1:

Suppose you know you'll need your data sorted by x for one procedure and by x and y for a different procedure. Just do one two-level sort from the beginning, instead of first sorting by x and then later sorting by x and y.

#### Example 2:

Use a `class` statement in procedures wherever possible instead of sorting and using a `by` statement. If your data are already sorted, however, a `by` statement is more efficient, since a `class` statement requires more memory.

### Tip 2: Sort Only What You Need To

#### Inefficient

```
proc sort data=old;
  by x;
run;

data new;
  set old (drop=i j k);
  where 10 < x < 20 ;
run;
```

#### Efficient

```
proc sort data=old(drop=i j k) out=new;
  by x;
  where 10 < x < 20;
run;
```

This eliminates unneeded variables before sorting instead of after.

### Tip 3: Use the `noequal`s Option

Normally, when sorting, SAS maintains the original order of the observations within each by group. Using the `noequal`s options specifies that observations within each by group do not have to stay in the same order as they were before sorting. If maintaining this order is not a concern for you, using the `noequal`s option will reduce CPU time.

# Write Clear and Easy-To-Understand Code

## Tip 1: Use a Header at the Beginning of a Program File

This can take whatever form you like, but should include basic documentation of what the program does, when it was created, last modified, etc. For example:

```
/******  
*                                                                 *  
*   Program Name: Example. sas                                 *  
*   Purpose: Shows some basic SAS operations                 *  
*   Produces Datasets: c:\example.sas7bdat                  *  
*   Created: May 20, 2000                                    *  
*   Last Modified: June 1, 2000                             *  
*   Author: Jackie Johnson                                   *  
*                                                                 *  
*****/
```

## Tip 2: Use Comments Throughout Your Program

Writing comments is very important, as you or someone else may want to go back to one of your programs someday and try to understand what it was doing and why it was doing it. Comments can be of two forms:

```
/*Put your comment here*/  
  
*Put your comment here;
```

## Tip 3: Use Sensible and Consistent Variable Names

Use variable names that in some way describe the data they contain. Using non-descriptive variable names such as `var1`, `var2`, etc. makes understanding the purpose of the program very hard, especially when multiple programmers may be working on the same project. Consistently using the same types of names for certain types of variables, especially when writing SAS programs frequently, will also increase readability.

## Tip 4: Separate Program Statements With `Run` Statements

Using a `run` statement after every `data` or `proc` step assures that comments about these in the SAS log will appear directly after the code they refer to, making the log much easier to read. Additionally, you can use `skip` and `page` statements to make sections even further stand out.

### **Tip 5: Write Expressions Clearly**

Write statements in the clearest and most easy-to-read format possible. Indentation, extra spaces, consistent capitalization, and added parentheses usually increase readability.

### **Tip 6: Specify Dataset Names in Procedures**

Without specifying a dataset in a procedure, the procedure is performed on the last dataset used. Although this may be fine, specifying a dataset name in a procedure makes it easy to see what dataset the procedure is working on, especially in long programs where statements for this dataset may be pages before the procedure where it is called.

### **Tip 7: Add Meaningful Titles to Procedures Producing Output**

Distinguishing between multiple pages of output is hard, especially if you run the same procedure many times on different datasets or different observations. Adding titles to these is an easy way to remedy this problem.