Paper 209-2007

The Most Important Efficiency Techniques

Bob Virgile, Robert Virgile Associates, Inc.

Overview

Even the title of this presentation raises some questions. What makes a program efficient? What makes one technique more important than another?

Most approaches to efficiency focus on speeding up the program. However, the programmer should consider much more than that. Is the program easy to understand and maintain? Does it require vast amounts of other resources (disk space, memory, tape drives to name a few). Does the analyst have to search through pages and pages of output to locate a few key numbers? How much time can you afford to learn and apply new techniques?

While much of this presentation will help speed up programs, it is still up to you to decide when and where it makes sense to incorporate the various techniques into your programs. One worthwhile strategy is to start with a few good techniques and make them a habit. As you add to your list of good habits over time, it becomes easier and easier to automatically use better techniques when you program.

Test the Outcome

Relative speed can change depending on the release of the software, the operating system, or characteristics of the data. From time to time, test the speed of alternative techniques using your current hardware and software. To support this effort, you do *not* have to locate suitable data sets with the proper size and data elements. Instead, create them! Here is one example:

```
data testdata;
   do v1=1 to 1000000;
      output;
   end;
   retain v2-v50 0 v51-v100 'ABC';
run;
```

Alternatively, don't even bother to create data. Just test within a do loop:

```
data _null_;
   length newvar $ 1;
   name='Bob';
   do i=1 to 500000;
      newvar = substr(name,1,1);
   end;
run;
```

```
data _null_;
   length newvar $ 1;
   name='Bob';
   do i=1 to 500000;
      newvar = name;
   end;
run;
```

Let's take a look at some common programming tasks, and methods that might make a program faster or slower.

Reading Raw Data

When reading from raw data, some measures are straightforward. For example, save permanent SAS® data sets, because reading from raw data takes longer. Also, read in just the needed variables. But how does this seemingly straightforward program perform extra work?

```
data retired_males;
   infile rawdata;
   input LastName  $  1-20
         FirstName $ 21-35
         Street    $ 36-55
         City      $ 56-75
         ZipCode   $ 76-80
         Gender    $ 81
         Age         82-84;
   if gender='M' and age >= 65;
run;
```

Presumably, subsetting deletes most observations. After all, males over 65 comprise just a fraction of a typical population. So for most observations, the program reads the first five variables unnecessarily. This replacement strategy would run faster (possibly much faster):

```
data retired_males;
   infile rawdata;
   input Gender    $ 81
         Age         82-84 @;
   if gender='M' and age >= 65;
   input LastName  $  1-20
         FirstName $ 21-35
         Street    $ 36-55
         City      $ 56-75
         ZipCode   $ 76-80;
run;
```

The replacement program reads just the variables needed to perform subsetting. The trailing @ holds each raw data line temporarily, in case the second input statement needs to read more variables from the same line. For the selected observations only, the data step reads in the remaining variables.

Finally, when reading from raw data, select informats wisely. These statements produce identical results:

```
input zipcode $ 48-52;
input @48 zipcode $5.;
```

Both statements read five characters, and left-hand justify whatever appears within those five characters. This statement is slightly different:

```
input @48 zipcode $char5.;
```

It reads the same five characters and copies them as is. By skipping the left-hand justification process, it performs less work.

Reading from SAS Data Sets

When working with SAS data sets, use keep and drop to control which variables get processed. The comment statement below is identical to the keep= data set option:

```
data bigwigs (keep=id nextyear);
   set employees;
   nextyear = salary * 1.05 + bonus;
```

```
      if nextyear > 100000;
    * keep id nextyear;
run;
```

However, adding keep= or drop= to the set statement provides additional savings by limiting which variables get read in:

```
data bigwigs (keep=id nextyear);
   set employees (keep=id salary bonus);
   nextyear = salary * 1.05 + bonus;
   if nextyear > 100000;
run;
```

The same principal applies when sorting data. The first program sorts all the variables, because keep= applies to the output data set:

```
proc sort data=huge out=tiny (keep=var1 var2 var3);
   by var1;
run;
```

But this variation sorts just three variables:

```
proc sort data=huge (keep=var1 var2 var3) out=tiny;
   by var1;
run;
```

Testing on a Sample

While it is good practice to test your programs on a sample of the data, issues will arise. For example, this program destroys a permanent SAS data set:

```
options obs=50;
proc sort data=huge.file;
   by id;
run;
```

And this program generates an error message on older releases of the software:

```
options obs=50;
data retirees;
   set all_workers;
   where age > 65;
run;
```

The safest approach is to limit observations at particular spots in the program:

```
data subset;
   set all_records (obs=50);
run;
```

When subsetting from raw data, the number of selected records is difficult to predict:

```
options obs=5000;
data first_50;
```

```
   infile rawdata;
   input name $10.;
   if name='Bob';
run;
```

The subset might contain anywhere from 0 to 5,000 observations. Instead of guessing, modify the data step to select a subset with exactly the right number of observations:

```
data first_50;
   infile rawdata;
   input name $10.;
   if name='Bob';
   found + 1;
   output;
   if found=50 then stop;
   drop found;
run;
```

Subsetting Considerations

When subsetting observations, where is usually faster than if:

```
data just_bob;
   set everybody;
   if name='Bob';
run;
```

```
data just_bob;
   set everybody;
   where name='Bob';
run;
```

Results will vary, depending on the release of the software, the percentage of observations selected, and the number of variables in the data set.

When subsetting with if, subset before manipulating the data. The first data step uses extra CPU time, calculating three variables before deleting some observations:

```
data just_bob;
   set everybody;
   hwratio = height / weight;
   kg = lb * 2.2;
   feet = inches / 12;
   if name='Bob';
run;
```

```
data just_bob;
   set everybody;
   hwratio = height / weight;
   kg = lb * 2.2;
   feet = inches / 12;
   where name='Bob';
run;
```

Procedures can use where, eliminating the need for a data step to subset observations. The first program contains an extra data step:

```
data highrisk;
   set patients;
   where cholesterol > 240;
run;
proc means data=highrisk;
   var height weight pulse;
run;
```

```
proc means data=patients;
   where cholesterol > 240;
   var height weight pulse;
run;
```

The second program lets the procedure perform the subsetting. However, if five procedures each need the same subset,

create the subset. The same where statement in multiple procedures would force each procedure to subset the observations. This principle applies even when the subset is based on variables rather than observations.

File Handling Considerations

Eliminate extra passes through the data. Some of the examples are simple ones:

- Don't sort if you don't have to. When a procedure (such as proc freq) is not using a by statement, it does not require sorted data.
- Don't sort data sets that are already sorted.
- Process permanent SAS data sets. There is no need to copy a data set to the work area, just to run a procedure against it.
- Eliminate data steps by combining all data manipulation into a single step.
- Create multiple output data sets in a single data step.
- Append using proc append, not a data step.
- If a data step is needed to create analysis variables, perhaps that data step can also perform the analysis.
- Sort once, in the most detailed order.
- Store data in the most commonly needed sorted order. If sorting into a different order, use out= to avoid replacing the original data set.

Here is an example where the output data set is still in sorted order. Once you think about it, there is no need to sort afterwards:

```
data no_outliers;
   set everyone;
   by id amount;
   if first.id or last.id then delete;
run;
```

Some Sorting Options

Consider the order of observations generated by these sorts. The outcome is identical, regardless of which program performs the sorting:

```
proc sort data=sales;
   by pop;
run;
proc sort data=sales;
   by state;
run;
```

```
proc sort data=sales;
   by state pop;
run;
```

The middle sort maintains the order within each state: from lowest to highest pop. What if you don't care about the order within each state? The software still maintains the incoming order for each state. That takes some work, which can sometimes be avoided. The noequals option is saying that the order within each by group does not matter:

```
proc sort data=sales noequals;
   by state;
run;
```

Some operating systems ignore the noequals option. In that case, the program still runs, as if the option had been omitted.

Consider another option that modifies the behavior of proc sort:

```
proc sort data=sales tagsort;
   by state;
run;
```

The normal action for proc sort is to sort the entire observation, including all the variables. The tagsort option requests a different method. It takes the value of state, and appends a “tag” (a numeric value holding the observation number in the incoming data); it then sorts just state and the tag. Once all the sorting has been completed, it uses the tag to retrieve the remaining variables from the original data set. Speed will vary (and could even increase), depending on the operating system and release of the software. But the sorting process uses a lot less disk space.

Some Miscellaneous Considerations

These two statements are different:

```
data _null_;
data;
```

The first statement creates no output data set, saving the CPU time needed to output as well as the storage space. The second statement says, “I’m too lazy to name the data set.” The software will supply the name.

When using proc tabulate to create a table, it often takes a few tries to format the table properly. As an approach, create a summary data set holding the final numbers. Then experiment with proc tabulate on the summary data set.

Summarizing Data

When summarizing data, learn to use a class statement rather than a by statement. This program processes the data twice:

```
proc sort data=golfers;
   by id;
run;
proc means data=golfers;
   by id;
   var score;
run;
```

But this program processes the data only once:

```
proc means data=golfers;
   class id;
   var score;
run;
```

There are enough details to learn. For example, any output data sets will be different when switching from a by statement to a class statement. However, the savings make it worth the effort.

To group data when summarizing, use a format rather than a data step. The long way would use a data step to modify the incoming data:

```
data groups;
   set golfers;
   length category $ 6;
   if handicap < 0 then category='pro';
   else if handicap=0 then category='par';
   else category='duffer';
```

```
run;
```

Then feed the modified data into procedures:

```
proc means data=groups;
   class category;
   var score;
run;
proc freq data=groups;
   tables category;
run;
```

The shorter way would create a format first:

```
proc format;
   value handcat low - <0 = 'pro'
                        0 = 'par'
                  0 - high = 'duffer';
run;
```

Then apply that format in later procedures:

```
proc means data=golfers;
   class handicap;
   format handicap handcat.;
   var score;
run;
proc freq data=golfers;
   tables handicap;
   format handicap handcat.;
run;
```

The difference is that the initial step (a format vs. a data step) no longer needs to process the entire data set. (Also note that there may be other differences if careless programming ignores the possibility of missing values in the original data set.)

Summarize a Summary

With proper planning, many programs can avoid processing a large data set. However, the planning process involves analyzing what numbers will be needed, and saving those numbers once in a summary data set. Then later programs can access the summary.

Here are a few examples of summarizing a summary data set. Begin by creating a summary:

```
proc means data=golfers noprint nway;
   class golfer course;
   var score;
   output out=totals sum=sum n=n;
run;
```

Or:

```
proc freq data=golfers noprint;
   tables golfer*course / out=counts;
```

```
run;
```

Then in dozens of subsequent programs, process the summary instead of the larger original data set. Some possibilities:

```
proc means data=totals noprint;
   by golfer;
   var sum n;
   output out=bygolfer sum=;
run;
proc freq data=counts;
   tables course golfer;
run;
proc means data=counts sum;
   class course;
   var count;
run;
```

Regardless of what these procedures are trying to accomplish, the point is that they run against a small data set instead of against the original data. The existence of summary data sets permits subsequent programs to run much faster.

Data Manipulation

It's hard to slow down a program significantly, via data manipulation. But a couple of principles are worth noting.

Functions (such as index, today(), and substr) are wonderful tools when needed. But they come with a cost. As a general rule, do not use the exact same function twice. Instead, use it once and save the result into a data step variable. Then use the variable multiple times as needed. For a simple example, consider the today() function:

```
now = today();
```

Instead of executing the function for each observation, execute it once and save the results:

```
if _n_=1 then now = today();
retain now;
```

Or, get fancy and use macro language:

```
retain now "&sysdate9"d;
```

Occasionally, functions can be replaced with faster tools. Consider substr, for example:

```
length title $ 4;
title = substr(fullname, 1, 4);
```

Since these statements copy the beginning of fullname, try this replacement:

```
length title $ 4;
title = fullname;
```

Even comparisons can make use of this principle. The long way:

```
if substr(lastname, 1, 1) = 'R';
```

When comparing the beginning of character strings, use the colon modifier to shorten the number of characters used in making the comparison:

```
if lastname =: 'R';
```

Since the letter R contains one character, the comparison (is lastname equal to R) is based on the first character of each string.

Mathematical operations run quickly, except when performed on missing values or dividing by zero. If the variable q1 is often missing, it may be faster to check for missing values:

```
if (q1 > .) then total = q1 + q2 + q3 + q4;
```

In similar fashion, check for division by zero:

```
if (weight ne 0) then hwratio=height/weight;
```

In this case, it is actually possible to check for both zero and missing values at the same time:

```
if weight then hwratio=height/weight;
```

Numeric values (such as the value of weight) are taken to be false when zero or missing, but true otherwise.

Storage Space

Some methods to save on storage space are simple:

- Save just the necessary variables.
- Compress SAS data sets.
- Use the minimum length needed for each variable.
- Save a code, print a format.

For example, the data set can save a company symbol. But when it comes time to print, it would be possible to print the company name by applying a format:

```
proc format;
   value $symbol  'GE'='General Electric'
                  'GM'='General Motors'
                   'F'='Ford';
run;
```

In some cases, this approach makes it worth the effort to learn proc report instead of proc print. Proc report can print the same variable twice, but using different formats for each column.

When using functions to create a new variable, the length assigned may not be the length you expect. Consider substr, for example:

```
letter = substr(name, 1, 1);
```

If this statement creates letter, it assigns letter the same length as name. A handful of other functions work the same way, including compress, left, right, reverse, translate, trim, and upcase.

Some functions are worse. If used to create a new variable, they define the new variable as being 200 characters long (the

maximum possible length under earlier releases of the software). Here's one example:

```
underline = repeat('_', 20);
```

Some other functions that use this length of 200: scan and symget.

Compressing a SAS data set is easy. Add any of these statements:

```
options compress=yes;
options compress=char;
options compress=binary;
```

The first two methods are identical, compressing repetitions of the same character. Those methods are better at compressing character strings. The third compresses patterns of bytes. It is better at compressing numeric variables, since a series of variables with a missing value all contain the same pattern of bytes.

To improve the results of compression, current releases of the software automatically store variables internally in a modified order. All numeric variables get stored sequentially, and all character variables get stored sequentially.

Finally, learn to use views. Views are sets of instructions on how to extract data, rather than the data itself. Here is an example, where the bold section of the data statement instructs the software to save a view:

```
data golfers / view=golfers;
   infile rawdata;
   input golfer  $char24.
         score   5.
         course  $char20.;
run;
```

This data step is saving the instructions on how to retrieve observations from the raw data file. Use the view just as you would use a data set:

```
proc sort data=golfers out=perm.golfers;
   by course golfer score;
run;
```

This is a little more to it than that. For example, proc sort was forced to use the out= option. It would not be possible to replace the view with the data itself. However, there is no work space needed to store the data set golfers. The same would apply when analyzing the data instead of sorting it:

```
proc means data=golfers;
   class course;
   var score;
run;
```

Here the software reads observations from the raw data source, and feeds them directly to proc means. There is no need to save the observations in the work area.

Views can slow the program down. In particular, do not use the same view multiple times (at least not without realizing the consequences). Each use of the view requires the software to read in from the raw data. That can slow a program down considerably.

Contact Information

Comments, questions, and suggestions are always welcome:

Bob Virgile
Robert Virgile Associates, Inc.
44 Woburn Street
Lexington, MA 02420
(781) 862-4642
rvirgile@comcast.net