Writing Efficient SAS Codes

Chong-ho (Alex) Yu, Ph.D. (2012)

It is unfortunate that efficient computing gets less and less attention because CPUs become progressively faster and RAM and disk becomes progressively cheaper. For a small dataset, the difference between inefficient and efficient SAS codes may be un-noticeable. But for a large dataset, efficient computing is still very important.

There are two aspects of efficiency:

- **Efficient use of computing resources:** The definition of efficient computing is: Given that the results of two sets of program segments are equal, a better program is the one which consumes less computing resources, which include CPU cycles, RAM, and disk storage. In some situations, use of CPU and RAM and use of storage is in a negative relationship. For instance, if the index of data is stored in the hard disk, it takes less CPU and RAM to do sorting and parsing.
- **Efficient use of human resources:** If two sets of codes consume equal amount of computing resources and produce the same results, but one set requires less human efforts (typing, modification, maintenance...etc), that one is considered more efficient.

Usually a more compact program requires less computing power and may even require less human resources. For instance, Novell Netware has 10-million lines of source code whereas Windows 2000 has 40 to 45-million lines. Even if the two network operating systems carry the same features, the one with less lines of codes is considered more desirable.

Besides shortening the program, there are other ways to achieve efficient computing. This write-up will illustrate efficient computing with examples of SAS codes.

Logical branching and comparison

The first example is conditional branching. When a blocking factor such as "age" is used in computing a ANOVA model, conditioning branching should be employed. Compare the following two sets of codes:

```
If age <= 10
        then group = "child    ";
If age => 11 and age <= 19
        then group = "teenager   ";
If age => 20 and age <= 29
        then group = "young adult";
If age => 30 and age <= 45
        then group = "adult      ";
If age => 46 and age <= 59
        then group = "middle age ";
If age => 60
        then group = "senior    ";
```

```
If age <= 10
        then group = "child    ";
else if age <= 19
        then group = "teenager   ";
else if age <= 29
        then group = "young adult";
else if age <= 45
        then group = "adult      ";
else if age <= 59
        then group = "middle age ";
else group = "senior    ";
```

Which set of source codes is more efficient? The answer is: the one on the right hand side, which is modified by William Griner.

- The program segment on the left uses "if" instead of "else if" after the first if-then statement. For each if-then statement, SAS must parse the entire dataset to classify the subjects into proper age groups. In the other program, SAS put aside all children after processing the first if-then statement and scan only the rest of the data. After processing the second if-then statement, SAS ignores all children and teenagers, and only look for young adults in the remaining data, and so forth.

The following code, which is suggested by my coworker, Eldon Norton, is also more efficient than the upper left one.

```
select;
```

```
        when (age <=10)  group = "child          ";
        when (age <= 19)        group = "teenager   ";
        when (age <= 29)        group = "young adult";
        when (age <= 45)        group = "adult      ";
        when (age <= 59)        group = "middle age ";
        otherwise group = "senior    ";
end;
```

Overwriting same dataset and variable

The following may go against common sense. On some occasions, it is advisable to overwrite the same dataset and variables even if you have made changes on them. Doing so can release SAS from holding too many data on disk. Take a look at the two following pseudo codes:

```
Data one; infile "c:\data.txt";
        define variables;
Data two; set one;
        first program segment;
Data three; set two;
        second program segment;
Data four; set three;
        third program segment;
```

```
Data one; infile "c:\data.txt";
        define variables;
Data two; set one;
        first program segment;
        delete one;
Data one; set two;
        second program segment;
        delete one;
Data one; set three;
        third program segment;
```

The program segment on the left keeps all four datasets on disk all the time. But it may be unnecessary. If you will not reuse the temporary dataset, there is no need to keep all of them. Therefore, the program segment on the right deletes the same dataset after each data step.

Not only you should overwrite the same dataset, but also you should overwrite the same variables if necessary. Compare the following two sets of SAS codes:

```
array a{10} a1-a10;
array b{10} b1-b10;
        do i = 1 to 10;
        if a{i} = 7 then a{i} = 0;
        else if a{i} = 6 then b{i} = 1;
        else if a{i} = 5 then b{i} = 2;
        else if a{i} = 4 then b{i} = 3;
        else if a{i} = 3 then b{i} = 4;
        else if a{i} = 2 then b{i} = 5;
        else if a{i} = 1 then b{i} = 6;
        else if a{i} = 0 then b{i} = 7;
end;
```

```
array a{10} a1-a10
        do i = 1 to 10;
        if a{i} = 7 then a{i} = 0;
        else if a{i} = 6 then a{i} = 1;
        else if a{i} = 5 then a{i} = 2;
        else if a{i} = 4 then a{i} = 3;
        else if a{i} = 3 then a{i} = 4;
        else if a{i} = 2 then a{i} = 5;
        else if a{i} = 1 then a{i} = 6;
        else if a{i} = 0 then a{i} = 7;
end;
```

It is a common practice for researchers to recode the data. The preceding SAS codes just did that. Also, it is not unusual that people create a new set of variables to store recoded data as shown on the above left panel. Indeed, the program on the right panel is more efficient because it writes new data back to the original variables rather than creating new ones. By the first glance, the program on the right does not work. If the value "7" has been changed to "0" by the first if-then statement and the new value is written back to the variable, will the new value "0" be reverted to "7" by the last if-then statement? No, it is because here "else if" instead of "if" is used. After all "7"s are changed, they are put aside and unaffected by the subsequent "else if" statements. This is another reason why you should use "else-if" rather than "if."

Using numeric variable names

This tip is very simple. But it is often overlooked by many people. This simple tip is: Use numbers at the end of variable names rather than characters. Although either one does not make a difference in using CPU power, it does make a difference to human resources (typing and looking up field names)! Look at following two sets of variable definitions:

```
Data one; input
        Q1 Q1b Q1c Q1d Q1other
        Time_SH Time_Wk
        Com_Ex Web_Ex Res_Ex
        Q4a Q4b Q5c Q5d;
cards;
```

```
Data one;
        input Q1-Q16;
cards;
```

In SAS you can assign variables as "Q1-Q26," but you cannot assign variables as "Qa-Qz." If you use numeric variable names, you can be more efficient by saving time from typing and from matching the names on the hard copy and the variable names on the screen. When you have many variables, using character labels makes referencing extremely difficult. When I was an inexperienced SAS programmer many years ago, I coded a survey with over a few hundreds fields using character-based names. As a result...you know!

Further, when someday you want to rename the variables, using numeric names will be very convenient. For example, to rename Q1-Q100 as Question1-Question100, the code is: *data new(rename=(q1-q100 = question1-question100));* Last but not least, when you want to do arrray manipulation, you will find that it is much easier to assign an array like *array question(*) question1-question100;* On may argue that the data set might have a set of meaningful item ID and the programmer should not alter them arbitrarily. This issue can be easily resolved by assigning a set of temporary ID for data manipulation and analysis, and then replacing the temporary ID with the original ID in the final output (See Automation of changing ID).

Using a value list in a variable

This tip not only reduces the use of CPU and memory resources, but also saves yourself from tedious coding. The following two codes perform the same task. The one on the left repeats the same comparison using "or," but the one on the right simply puts a list of values into a variable. If you know the concept of array and list, you know processing a list or an array of data is faster than processing data one by one. In addition, if you reuse the same code over and over, you can assign a macro variable called "delete_list," such as %LET delete_list = %str("Tom", "Peter", "Mary", "Alex", "Jane", "Louis"); Next time you can simply update the list in macro.

```
If name = "Tom" or
  name = "Peter" or
  name = "Mary" or
  name = "Alex" or
  name = "Jane" or
  name = "Louis" then delete;
```

```
If name in
        ("Tom","Peter","Mary",
        "Alex","Jane","Louis")
        then delete;

If name in (&delete_list) then delete;
```

PROC SQL vs. PROC SUMMARY

The following tip is provided by Eldon Norton. Once I wrote an inefficient SAS program to extract user log data from a web server. Eldon pointed out that to parse data, the **structural query language (SQL)** is more powerful than the regular data parsing method. For instance, the codes on the left panel uses three PROCs to rank webpage by the number of page access. The code on the right, which utilizes SQL, can perform the job in one PROC. Also, it is not necessary to create one more data set and thus it avoids further consuming computer resources.

```
data two; set one;
        count = 1;
        proc summary data=two;
                class page; var count;
                output out=new sum= ;
        proc sort; by descending count;
```

```
data one;
proc sql; select link,
        count(*) label=count from one
        group by link
        order by count;
        quit;
```

ARRAY vs. PROC TRANSPOSE

Many years ago I was a psychometrician of a corporation. Usually the test data are formatted in a tall structure as shown on the left panel below. In this hypothetical data set, there were five examinees and each one completed the test by taking all five items. Their scores were recorded in the column "itemscore." Many people used the SAS code as shown in the right panel below to transpose the data from a tall structure to a wide one. In other words, instead of putting the item scores into one single field, you have to spread the scores across many columns by the item ID numbers.

```
data b; set a; by userid itemid;

length i1-i5 $8;

array scores s1-s5;

array inames $ i1-i5;

retain s1-s5 i1-i5 n;

if first.userid then n = 1;
```

VIEWTABLE: Work.A

	userid	itemid	itemscore
1	1	1	1
2	1	2	0
3	1	3	1
4	1	4	1
5	1	5	0
6	2	1	1
7	2	2	1
8	2	3	0
9	2	4	0
10	2	5	1
11	3	1	0
12	3	2	1
13	3	3	1
14	3	4	1
15	3	5	1
16	4	1	0
17	4	2	1
18	4	3	1
19	4	4	0
20	4	5	1
21	5	1	0
22	5	2	0
23	5	3	0
24	5	4	1
25	5	5	0

```
scores(n) = itemscore;

inames(n) = itemid;

n = n+1;

if last.userid then output;

run;
```

It works fine as long as all examinees took all items. The perfect output is shown in the left panel below. However, what would happen if some examinees skipped parts of the exam but the database did not capture the missing data. For example, in the table as shown on the right panel below, User 3 skipped Item 3 and User 5 skipped the last item. Needless to say, the preceding SAS code is unable to yield the correct result. Specifically, the item scores would incorrectly shift the positions.

Perfect output

s1	s2	s3	s4	s5
1	0	1	1	0
1	1	0	0	1
0	1	1	1	1
0	1	1	0	1
0	0	0	1	0

Missing data due to skipping items

VIEWTABLE: Work.C

	userid	itemid	itemscore
1	1	1	1
2	1	2	0
3	1	3	1
4	1	4	1
5	1	5	0
6	2	1	1
7	2	2	1
8	2	3	0
9	2	4	0
10	2	5	1
11	3	1	0
12	3	2	1
13	3	4	1
14	3	5	1
15	4	1	0
16	4	2	1
17	4	3	1
18	4	4	0
19	4	5	1
20	5	1	0
21	5	2	0
22	5	3	0
23	5	4	1

The preceding code is very complicated and thus it is more error-prone. In addition, you need to know the total number of items in advance. Nevertheless, there is a more efficient solution that could return an accurate result regardless of the number of items and missing data. This is PROC TRANSPOSE as illustrated on the upper panel below. It is extremely simple yet it yields the right output (see the middle panel).

However, what would happen if the first examinee skipped some items, such as skipping Item 4? The good news is: you can still obtain the correct output using this approach. The drawback is: in this case because the first user didn't answer Item 4, in the first transpose there is no Item 4. Item 4 appears at the second transpose, but as a result the item order in the table is: itemid5, itemid4 (see the lower panel below). Nonetheless, it will not affect your data analysis as along as the right scores or missing scores attach to the right item ID numbers.

```
proc transpose data=c out=e prefix=itemid;
by userid;
id itemid;
var itemscore; run;
```

Correct output with missing values in the right positions

VIEWTABLE: Work.E

	userid	NAME OF FORMER VARIABLE	itemid1	itemid2	itemid3	itemid4	itemid5
1	1	itemscore	1	0	1	1	0
2	2	itemscore	1	1	0	0	1
3	3	itemscore	0	1	.	1	1
4	4	itemscore	0	1	1	0	1
5	5	itemscore	0	0	0	1	.

Still correct output with missing values but a different item order

VIEWTABLE: Work.E

	userid	NAME OF FORMER VARIABLE	itemid1	itemid2	itemid3	itemid5	itemid4
1	1	itemscore	1	0	1	0	.
2	2	itemscore	1	1	0	1	0
3	3	itemscore	0	1	.	1	1
4	4	itemscore	0	1	1	1	0
5	5	itemscore	0	0	0	.	1

Good luck! And happy SASing!

Navigation

[SAS tips contents]

[Computer write-ups contents]

[Simplified Navigation]

[Table of Contents]

[Search Engine]

[Contact]