

What is Efficient SAS Coding?

Philip R Holland, Holland Numerics Limited, Royston, Herts, UK

ABSTRACT

Coding efficiency is generally measured in CPU time, disk space or memory usage. This is perfectly reasonable for SAS code that will be submitted many more times than it will be updated. However, there are coding environments where SAS programs are being written for single production runs, and then adapted and updated for different production runs, e.g. in clinical development. In these cases a new measurement of maintenance time may be more important.

This paper discusses the choices that need to be made when coding efficiently in different types of SAS programs, each illustrated by coding examples.

INTRODUCTION

Traditionally coding efficiency has been measured using CPU time, memory usage or disk space usage. These measured are important when the programs are being used to process large amounts, i.e. gigabytes, of data, and are being run frequently, but are not likely to be frequently copied or updated.

SAS programs used in clinical trials are unlikely to be used to process large amounts of data, but are very likely to be updated and adapted for use in a series of trials requiring similar processing. Saving 50% of the CPU time, when the program only runs for 5 minutes, is not going to make a significant impact on the coding efficiency. However, a program that is difficult to maintain could add days, or even weeks, to the time needed to prepare the program for a new trial.

The following examples reflect my personal views on coding efficiency. In some cases the choice of an appropriate coding will depend on the programming experience within the SAS programming team, particularly their knowledge of SQL programming.

SPEED AND LOW MAINTENANCE

IF...THEN...ELSE and SELECT...WHEN constructs are examples of code that can be written to improve speed and maintenance time at the same time. In a simple case of an input dataset containing 3 possible values A-C for a variable, then the assignment of a new variable based on the value can be written in a number of ways. All 3 examples will generate exactly the same output dataset.

1. The following code is not efficient, as every IF condition will be applied to every record. However, for small input datasets, you may not notice any of the inherent inefficiency.

```
DATA new;
  SET old;
  IF oldvar = 'A' THEN newvar = 1;
  IF oldvar = 'B' THEN newvar = 2;
  IF oldvar = 'C' THEN newvar = 3;
RUN;
```

2. The following code is more efficient, as IF conditions will only be applied up to the condition that matches. However, for small input datasets, you may not notice the increased speed. Further improvements in speed can be achieved by ordering the IF conditions so that the most commonly used is placed at the top, but this may no prove to be worthwhile unless the data is already well-known and the most common value is very common.

```
DATA new;
  SET old;
  IF oldvar = 'A' THEN newvar = 1;
  ELSE IF oldvar = 'B' THEN newvar = 2;
  ELSE IF oldvar = 'C' THEN newvar = 3;
RUN;
```

3. The following code is comparable in efficiency to the code in (2), as WHEN conditions will only be applied up to the condition that matches. However, for small input datasets, you may not notice the increased speed. Again further improvements in speed can be achieved by ordering the WHEN conditions so that the most commonly used is placed at the top. In my opinion this construct will be easier to maintain, as all the lines have the same layout, so inserting or deleting lines can be carried out with a reduced risk of introducing syntax errors.

```

DATA new;
  SET old;
  SELECT (oldvar);
    WHEN ('A') newvar = 1;
    WHEN ('B') newvar = 2;
    WHEN ('C') newvar = 3;
    OTHERWISE;
  END;
RUN;

```

Extending conditional clauses to 10 or more conditions can require great care to avoid inefficient processing, especially if the input dataset is large. Inefficient maintenance can also be avoided, particularly if the conditional code has been enclosed in a DO...END construct, if the code is laid out with indents indicating the relative positions of each section of conditional code.

SPEED OR LOW MAINTENANCE – PART 1

Rewriting a Data Step merge with a PROC SQL join can help to reduce maintenance time, but may reduce the processing speed. In the sample code below I am merging 3 datasets using 2 different variables, then re-ordering the resulting dataset by another variable.

1. First a combination of PROC SORT and DATA steps, which can be quite efficient as far as processing is concerned, but is quite long and involved, as you have to sort the individual datasets prior to merging them together:

```

PROC SORT DATA = a OUT = a1;
  BY cat_b;
RUN;

PROC SORT DATA = b OUT = b1;
  BY cat_b;
RUN;

DATA a1_b1;
  MERGE a1 (IN = a) b1 (IN = b);
  BY cat_b;
  IF a AND b;
RUN;

PROC SORT DATA = a1_b1 OUT = a1_b11;
  BY cat_c;
RUN;

PROC SORT DATA = c OUT = c1;
  BY cat_c;
RUN;

DATA a1_b1_c1;
  MERGE a1_b11 (IN = ab) c1 (IN = c);
  BY cat_c;
  IF ab AND c;
RUN;

PROC SORT DATA = a1_b1_c1;
  BY cat_a cat_b cat_c;
RUN;

```

2. Second a single PROC SQL step, which does everything in this single step, including the final sort. When the input datasets are small to moderate in size, there is little difference in CPU time used by this and the code above, but very large input datasets can result in slower processing when using PROC SQL. Another obvious disadvantage is that, when combining 2 or more datasets with overlapping variables, you must list all the variables to be included in the output datasets. However, assuming that the SAS programming team has some experience with SQL programming, this program should be the easier to maintain:

```

PROC SQL;
  CREATE TABLE a_b_c AS
  SELECT a.cat_a
        ,b.cat_b
        ,c.cat_c
        ,a.num_a1
        ,a.num_a2
        ,b.num_b1
        ,b.num_b2
        ,c.num_c1
        ,c.num_c2
  FROM   a
        ,b
        ,c
  WHERE  a.cat_b = b.cat_b
        AND a.cat_c = c.cat_c
  ORDER BY
        a.cat_a
        ,b.cat_b
        ,c.cat_c
;
QUIT;

```

SPEED OR LOW MAINTENANCE – PART 2

Coding of simple merges is fairly straightforward using DATA or PROC SQL steps, but PROC SQL can be the number one choice when joining tables together based on a range of values, rather than a one-to-one match. In this example the code is being used to calculate the largest difference between records within 28 days of each other.

1. First a combination of PROC SORT, PROC TRANSPOSE and DATA steps, which can be quite efficient as far as processing is concerned, but is quite long and involved, as you have to use arrays to categorize all the individual pairs of records:

```

PROC SORT DATA = old OUT = temp;
  BY cat num;
RUN;

DATA temp;
  SET temp;
  BY cat;
  RETAIN order;
  IF FIRST.cat THEN order = 1;
  ELSE order + 1;
RUN;

PROC TRANSPOSE DATA = temp OUT = num PREFIX = num;
  BY cat;
  VAR num;
  ID order;
RUN;

PROC TRANSPOSE DATA = temp OUT = val PREFIX = val;
  BY cat;
  VAR val;
  ID order;
RUN;

DATA all (DROP = _:);
  MERGE num val;
  BY cat;
RUN;

```

```

DATA new (KEEP = cat maxval);
  SET all;
  BY cat;
  ARRAY num num;;
  ARRAY val val;;
  ARRAY test test1-test50;
  maxval = .;
  reset = 1;
  DO i = 1 TO DIM(num);
    DO j = i+1 TO DIM(num);
      IF num(j) - num(i) LE 28 AND ROUND(val(j) - val(i), .0001) GT maxval
        THEN maxval = ROUND(val(j) - val(i), .0001);
    END;
  END;
  IF LAST.cat THEN OUTPUT;
RUN;

```

2. Second just 2 PROC SQL steps, one to join the input dataset with itself to generate all possible combinations of 1 to 28 day gaps, and a second PROC SQL step to find the largest value difference. Again, when the input datasets are small to moderate in size, there is little difference in CPU time used by this and the code above, but very large input datasets can result in slower processing when using PROC SQL. However, assuming that the SAS programming team has some experience with SQL programming, this program should be much easier to maintain:

```

PROC SQL;
  CREATE TABLE temp AS
  SELECT b1.cat
         ,b1.num
         ,MAX(b2.val - b1.val) AS maxval
  FROM   old b1
  LEFT JOIN
         old b2
  ON     b1.cat = b2.cat
        AND (b2.num - b1.num) BETWEEN 1 AND 28
  GROUP BY
         b1.cat
         ,b1.num
  ;
QUIT;

PROC SQL;
  CREATE TABLE new AS
  SELECT cat
         ,MAX(maxval) AS maxval
  FROM   temp
  GROUP BY
         cat
  ;
QUIT;

```

PERSONAL PREFERENCES

Every SAS programming team has its own “standard” reporting procedure, usually PROC REPORT or PROC TABULATE. In terms of processing time there is little to choose between them, but strangely combining PROC SUMMARY with PROC PRINT can create very acceptable tables in less processing time. As far as maintenance time is concerned, this is probably a matter of what you are used to.

PhUSE 2006

1. PROC REPORT is compact and quite easy to maintain, with the order of the report columns determined by the COLUMN statement:

```
PROC REPORT DATA = old NOWD;
  TITLE "Report";
  COLUMN cat_a cat_b = n_b cat_b = pct_b num val;
  DEFINE cat_a / GROUP 'Category';
  DEFINE n_b / SUM FORMAT = 8. 'N b';
  DEFINE pct_b / MEAN FORMAT = PERCENT8. 'Pct b';
  DEFINE num / MEAN FORMAT = 8.1 'Mean num';
  DEFINE val / MEDIAN FORMAT = 8.1 'Median val';
RUN;
```

2. PROC TABULATE has a more complex syntax, but is easier to use when you need to include multiple statistics for a single variable. All you have to remember is the syntax follows the simple rule, **[[[Page,] Row,] Column]:**

```
PROC TABULATE DATA = old;
  TITLE "Tabulate";
  CLASS cat_a;
  VAR cat_b num val;
  TABLE cat_a = 'Category'
    , (cat_b = ' ' * (SUM = 'N b' * F = 8. MEAN = 'Pct b' * F = PERCENT8.)
      num = ' ' * MEAN = 'Mean num' * F = 8.1
      val = ' ' * MEDIAN = 'Median val' * F = 8.1
    )
  ;
RUN;
```

3. PROC PRINT is often ignored, but with PROC SUMMARY it makes a useful alternative, with a simple syntax and fast processing:

```
PROC SUMMARY DATA = old NWAY;
  CLASS cat_a;
  VAR cat_b num val;
  OUTPUT OUT = temp SUM(cat_b) = n_b
                    MEAN(cat_b num) = pct_b num
                    MEDIAN(val) =
  ;
RUN;

PROC PRINT DATA = temp LABEL;
  TITLE "SUMMARY + PRINT";
  VAR cat_a n_b pct_b num val;
  LABEL cat_a = 'Category'
        n_b = 'N b'
        pct_b = 'Pct b'
        num = 'Mean num'
        val = 'Median val'
  ;
  FORMAT n_b 8.
         pct_b PERCENT8.
         num val 8.1
  ;
RUN;
```

REDUCING MAINTENANCE

In the following examples the PROC SQL code is exactly the same, but, in my opinion, the coding layout (3) will be easier to maintain, as all the lines have the same layout, so inserting or deleting lines can be carried out with a reduced risk of introducing syntax errors.

1. "Prose" layout, which is quick to write, but can be a nightmare to maintain:

```
PROC SQL;
  CREATE TABLE new AS SELECT a.col1, b.col2, a.col3 FROM a, b
  WHERE a.col1 = b.col1 AND a.col2 = b.col2;
QUIT;
```

2. "Split" layout, which is easier to read, but inserting and deleting lines of code can introduce syntax errors:

```
PROC SQL;  
  CREATE TABLE new AS  
    SELECT a.col1,  
           b.col2,  
           a.col3  
  FROM a,  
       b  
 WHERE a.col1 = b.col1 AND  
       a.col2 = b.col2;  
QUIT;
```

3. "Comma first" layout, which is easier to read, but inserting and deleting lines of code can be carried out quite safely. Note that highlighted lines are always written with commas and operators at the beginning, so that they can be safely duplicated or deleted:

```
PROC SQL;  
  CREATE TABLE new AS  
    SELECT a.col1  
           ,b.col2  
           ,a.col3  
  FROM a  
       ,b  
 WHERE a.col1 = b.col1  
       AND a.col2 = b.col2  
;  
QUIT;
```

MY RULES OF THUMB

- Any section of code used to create a single variable that cannot be printed on a single side of A4 paper is too complex.
- If you are processing a small amount of data, then saving 50% processing time by spending 50% more development time is not efficient coding.
- High speed and low maintenance time is efficient coding.
- Low speed and high maintenance time is inefficient coding.
- The efficiency of coding with high speed and high maintenance time, or low speed and low maintenance time, will depend on how often the program will be submitted.

REFERENCES

- SAS Training Course: SAS Programming III: Advanced Techniques.

CONTACT INFORMATION

The author is a consultant for Holland Numerics Ltd and can be contacted at the following address:

address:	Philip R Holland Holland Numerics Ltd 94 Green Drift Royston Herts. SG8 5BT UK
e-mail:	phil@hollandnumerics.com
web:	www.hollandnumerics.com
tel. (mobile):	+44-(0)7714-279085

This paper and associated sample SAS code can be downloaded from the Holland Numerics Ltd web site at:
www.hollandnumerics.com/SASPAPER.HTM

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.