Effecting Efficiency Effortlessly

Daniel Carden, Quanticate, Manchester, United Kingdom

ABSTRACT

There are always multiple ways of doing things in SAS to produce required outputs. Often the way that ensures the quickest program run time is the best, as this will save costs in time and money. However, efficient techniques have to be well understood and used correctly to avoid error. It is also crucial that comments explain how a technique has been used so that another programmer can know exactly what is going on.

The aim of this paper is to effect efficiency effortlessly, by using a number of techniques including SAS views, efficient code structuring, WHERE statements, use of a Skip Macro in place of commenting-out code and creating Format Libraries in order to save CPU time. These techniques can save hours of real time, and when properly understood, used and narrated by a programmer they can do so without impacting on the overall "understandability" of the code.

INTRODUCTION

Programs should run as fast as possible in order to minimise costs. However, if time efficient techniques are to be used, these must be well explained so that code changes can be easily made and other programmers can pick up the code at a later date. The aim of this paper is to go through a range of efficient coding techniques to make programs run faster with a minimum of effort.

SAS VIEWS:

Since SAS Version 6 was introduced, SAS views have offered an alternative SAS data set to SAS data files. Whereas SAS data files contain actual data, SAS data views refer to data stored in other file structures such as ORACLE tables. There are three types of SAS data view:

- **DATA step views** are a type of data step program.

- **PROC SQL views** are stored query expressions that read data values from their underlying files, which can include SAS data files, SAS/ACCESS views, DATA step views, other PROC SQL views, or relational database data.

- **SAS/ACCESS views** (also called view descriptors) describe data that is stored in DBMS (Database Management System) tables.

This paper will only consider DATA step views as shown in Example 1:

Example 1:

```
data labs / view = labs;
  set labsdata;
  gender = sex;
  label gender = 'Gender Type';
  mid = (lowrang + hirang)/2;
run;

data labs2;
  set labs;
run;
```

In the first data step, the SAS view *labs* is created from the data set *labsdata* in the Work library. No SAS data file is created from this step. When the data file *labs2* is created in the second step, the code refers to the SAS view *labs* which then refers to the data in *labsdata*. *Labs* has no data and is an intermediate step between *labsdata* and *labs2*. Take note of the syntax of data <name> / view = <name>.

SAS views do not have data because SAS data files consist of two parts: a descriptor portion and a data portion. The descriptor portion contains the name and properties of the data set, such as when it was created, and the number of observations and variables. The data portion contains the data values. A SAS data file stores descriptor information and data values together. In contrast a SAS data view defines a virtual data set. It has the information required to access data values and is stored separately from the data values. It is a stored, compiled data step program.

SAS views can save real time. However, they do not do this by reducing the time the Central Processing Unit spends performing the operations you assign (CPU time). If anything they increase CPU time slightly. What SAS views avoid is the writing and reading back of temporary data sets, thereby cutting I/O time. I/O time is the time the computer spends on two tasks, input and output. Input refers to the moving of data from storage areas such as disks and tapes into memory. Output refers to moving the results out of memory to storage or to a display device. Memory is the size of the work area that the CPU must devote to the operations in the program. Since it is by reducing I/O time that SAS views reduce real time, they are best used in data steps that create intermediate temporary data files and have real execution times which greatly exceed associated CPU times.

To investigate how SAS views can save processing time when execution times exceed CPU time, the code from Example 1 was run seven times using a standard method (*data labs;*) and using the SAS view method. *Labsdata* has just over 340 000 observations so is an ideal test data set. The average run time using the standard method was 49.3 seconds, whereas using SAS views the average time was 30.0 seconds on a Windows XP operating system. The reason for this time saving is because of *labs*. Using the standard method involves two steps. All the data from *labsdata* is rewritten to *labs*, which is then reread by *labs2*. The SAS view method cuts I/O time by not creating the *labs* data file. When *labs2* is run the data is effectively read in one step from *labsdata* to *labs2*.

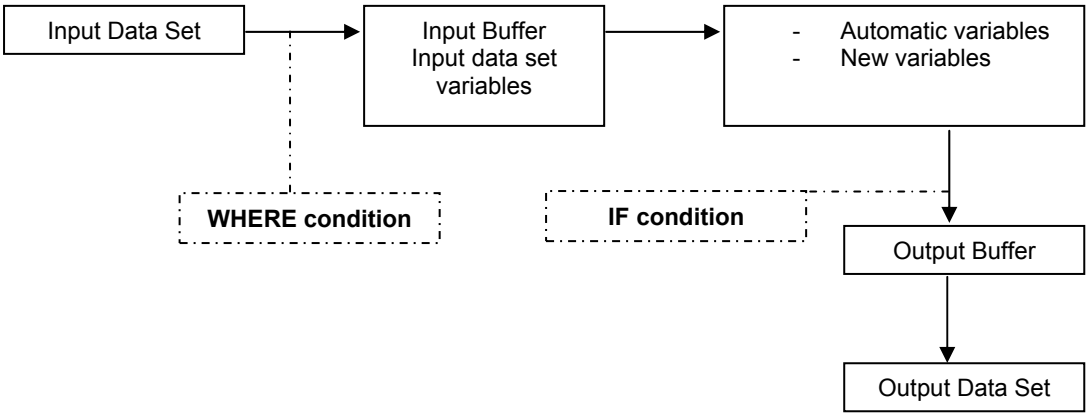
The advantage of the SAS view method is the speed of the run time. It also reduces the maximum disk space requirements for a given job by reducing the redundant copies of data required to be held on disk at any moment. However, there are a couple of disadvantages. SAS views do not show errors in the log as consistently as in standard data steps. For example, if 100 is added to a character variable in a SAS view code no mention is made of it in the log. Also, SAS views cannot overwrite or be overwritten by any data set of the same name, limiting flexibility. SAS views are best used when a large data set needs to be referenced several times in a piece of code and when real run times are larger than CPU times. Derivations to create the large data set should be done in a standard data set, and then a SAS view created from this data step. This will save unnecessary processing time each time the data set is referenced later in the code.

WHERE STATEMENTS

When subsetting input data sets WHERE statements represent the most efficient method as compared with IF conditions. Figure 1 below shows in what stage of the data set process WHERE and IF operate,

assuming that WHERE is applied to the set/merge statement in the data step rather than applied to the data statement:

Figure 1:



When applied to set/merge WHERE is more efficient because it acts on the data set at an earlier stage than IF. For the same reason when DROP is used it is more efficient to apply it to set/merge rather than to the data statement.

CODE STRUCTURING

Code structuring is crucial to reducing program run times. Minimising the number of data steps is one way to do this, even if there is a trade-off in terms of readability of the code:

Example 2:

Two data step method	**One data step method**
19 data labs; 20 set labsdata; 21 where obssd^=0; 22 run; NOTE: There were 319452 observations read from the data set WORK.LABSDATA. WHERE obssd not = 0; NOTE: The data set WORK.LABS has 319452 observations and 39 variables. NOTE: DATA statement used: real time 22.91 seconds cpu time 0.98 seconds 23 24 25 proc sort data = labs out = labs2; 26 by pt invsite; 27 run; NOTE: There were 319452 observations read from the data set WORK.LABS. NOTE: The data set WORK.LABS2 has 319452 observations and 39 variables. NOTE: PROCEDURE SORT used: real time 1:00.63 cpu time 2.78 seconds	30 proc sort data = labsdata (where = (obssd^=0)) out = labs; 31 by pt invsite; 32 run; NOTE: There were 319452 observations read from the data set WORK.LABSDATA. WHERE obssd not = 0; NOTE: The data set WORK.LABS has 319452 observations and 39 variables. NOTE: PROCEDURE SORT used: real time 57.39 seconds cpu time 1.73 seconds
Total CPU run time = 0.98s + 2.78s = 3.76 s	**Total CPU run time = 1.73 s**
Total real run time = 1m0.6s + 22.9s = 1m23.5s	**Total real run time = 57.4s**

The above example is exaggerated because the 2nd data step which creates *labs2* on this occasion took longer than the same sort from *labsdata* to *labs*. However, it is clear that the combination of run times from the two data step method will tend to be slower than the run time of the one step method.

Another way to reduce run times is to ensure that macros are invoked only when needed. The example below shows 2 ways of creating the same output(s) *vitlab&n*.

Example 3:

Method 1	**Method 2**

`%macro labvital (n=, where=);`
 	
`proc sort data = rawdata.vitals out = vitals nodupkey; by pt;`
`run;`
 	
`data labs&n;`
`  set labsdata;`
`  where &where;`
`  mid = (lowrang + hirang)/2;`
`  if hirang > 0 then percent = (lowrang /hirang) * 100;`
 	
`run;`
 	
`data vitlab&n;`
`  merge labs&n vitals; by pt;`
`run;`
 	
`%mend;`
 	
`%labvital (n= 1, where= lvaluen>0.5);`
`%labvital (n= 2, where= lvaluen>1);`
`%labvital (n= 3, where= lvaluen>1.5);`
`%labvital (n= 4, where= lvaluen>2);`
`%labvital (n= 5, where= lvaluen>2.5);`
`%labvital (n= 6, where= lvaluen>3);`
	`proc sort data = rawdata.vitals out = vitals nodupkey; by pt;`
	`run;`
	
	`%macro labvital (n=, where=);`
	
	`data labs&n;`
	`  set labsdata;`
	`  where &where;`
	`  mid = (lowrang + hirang)/2;`
	`  if hirang > 0 then percent = (lowrang /hirang) * 100;`
	
	`run;`
	
	`data vitlab&n;`
	`  merge labs&n vitals; by pt;`
	`run;`
	
	`%mend;`
	
	`%labvital (n= 1, where= lvaluen>0.5);`
	`%labvital (n= 2, where= lvaluen>1);`
	`%labvital (n= 3, where= lvaluen>1.5);`
	`%labvital (n= 4, where= lvaluen>2);`
	`%labvital (n= 5, where= lvaluen>2.5);`
	`%labvital (n= 6, where= lvaluen>3);`

CPU run time = 2.31s	**CPU run time = 1.46s**
Total real run time = 72.51s	**Total real run time = 68.41s**

In Method 1, every time that the macro *labvital* is invoked the *vitals* data is output because it is included in the macro statement. Taking it outside the macro statement in Method 2 ensures that *vitals* is only output once, which will save time in proportion to the size of *vitals* and the number of times that the macro is invoked. Ideally *mid* and *percent* should also be derived before the macro *labvital* is invoked, thereby saving even more processing time.

SKIP MACRO

To create certain outputs, lengthy SAS code may need to be shortened by commenting out sections of it so that redundant code does not submit needlessly and waste run time. If only small amounts of code need commenting out then standard techniques like /* */ can be used but for longer pieces of code the Skip Macro is a more efficient technique. It is especially useful to comment out whole sections of code already broken up by /* */ as it excludes even the comments from being processed.

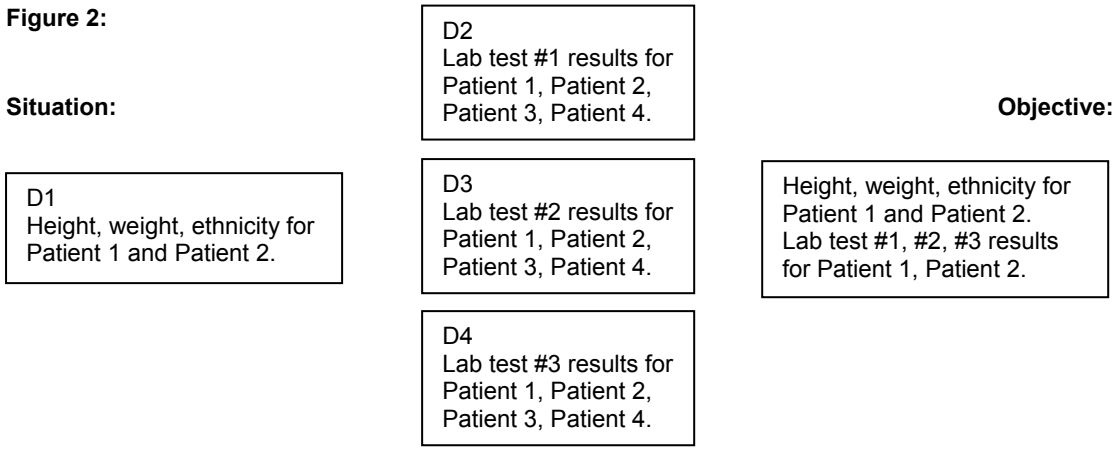
To use the Skip Macro place %macro skip; and %mend skip; above and below the code that you intend to comment out. The code between %macro skip and %mend skip will be ignored when the SAS program is run.

FORMAT LIBRARIES

Program efficiency is often related to restricting the amount of data being read in by SAS. One way to do this is to use an Index. A SAS Index is similar to a search function, allowing access to a subset of records from a large data set. Going through the intricacies of indices is outside the scope of this paper, and instead the less well-known topic of format libraries is introduced. Format libraries are similar to indices in that they subset large amounts of data, reducing run time. Their main use is in efficient merging.

Imagine the scenario where one data set D1 has to be merged with three others D2, D3, D4. Suppose that D1 has demographic data for a subset of the patient population, and this has to be merged with the results of three different lab tests (D2, D3, D4) carried out on the whole population. For explanation purposes, example data is included:

Figure 2:



A simple way to achieve this would be to set *D2*, *D3* and *D4* together, then do a merge of the form in Example 4, the IF condition outputting the desired records which are both in *a* (*D1*) and *b* (*D234*):

Example 4:

```
data D234;
    set D2 D3 D4;
    by subjid;
run;


data combine;
    merge D1 (in = a) D234 (in=b);
    by subjid;
    if a and b;
run;
```

The inefficiency of this approach arises from the set statement used to create *D234* and the merge statement to create *combine*. Outputting all records to *D234* rather than just the records required from *D2*, *D3* and *D4* is inefficient, as is reading those surplus records in the merge statement. Figure 2 and Example 4 show that the creation of *D234* and *combine* requires the lab test results for Patients 3 and 4 to be needlessly fed through SAS when only the lab test results of Patients 1 and 2 are needed.

A more efficient way of creating exactly the same output data set as *combine* is available by using a format library to subset *D234*. To store and use a format library, do the following (examples 5/6):

Example 5. Create a Format Library:

```
data D1;
  set rawdata.D1;
    start = subjid;
    fmtname = '$Fsubj';
    label = 'Y';
    type =  'C';
run;


proc format cntlin = D1;
```

After the dataset *D1* is created PROC format is used with the CNTLIN option to create the dataset into a Format Library. The CNTLIN option turns the data stored in the pre-format SAS data set (*D1*) to a SAS format and adds it to the format catalogue in the Work library.

To be made into a format library, a data set such as *D1* must have the following variables:

- *START: The value to format into a label (the KEY).
- FMTNAME: The name of the format being created, which can be anything except the name of a format which is already defined. When the KEY is character, FMTNAME must start with a $ just like any PROC FORMAT value.
- TYPE: Either character ('C') or numeric ('N') format.
- LABEL: The label given to the KEY variable. This can be anything, but must not be the first byte in the KEY.

*NB: There must not be any duplicates of the variable used as the KEY variable.

Once a format is created, it can be used anywhere in the program for selecting matches to the KEY.

Example 6. Apply the Format Library:

```
data D234;
    set D2 D3 D4;
    by subjid;
    if put (subjid,$Fsubj.)='Y';
run;


data combine;
    merge D1 D234;
    by subjid;
run;
```

In the first data step, the IF condition looks for subjects who have the format *Fsubj* equal to *'Y'* and only outputs those subjects to *D234* saving output time. Input time is then saved in the *combine* data step by *D234* containing only the relevant data of *D1* subjects. Note that a WHERE clause creates errors when applying the library, so IF is used instead. Using this method on a *D1* data set with ~ 230 000 observations and 40 variables, and a *D2/D3/D4* data set each having ~ 680 000 observations and 40 variables in Windows XP, the time differences from each method were as follows:

Figure 3:

Method	CPU Time	Real Time
Format Library	11.24 seconds	2 minutes 37 seconds
Standard Approach	12.25 seconds	5 minutes 53 seconds

As is clear from Figure 3, the format libraries method can have a sizeable impact on run times when working with large data sets. It can lead to further time savings if the format library is used again later in the code, or if time-intensive calculations have to be performed on the set together data set (e.g. *D234*).

CONCLUSION

The purpose of this paper has been to explain a variety of useful techniques to program efficiently. It is hoped that the explanations provided are sufficiently clear so that implementing the techniques: SAS views, WHERE statements, Code Structuring, the Skip Macro and Format Libraries will indeed prove effortless.

It is inevitable that as technology continues to advance more and more data will be generated not just in the pharmaceutical industry but beyond, and that programmers will have to find innovative new ways to handle it. It is hoped this paper goes some way to alleviating this issue of too much data, too little time and has provided you with some inspiration to improve your future SAS programs.