PhUSE EU Connect 2018

Paper CT06

Time-Management: Programming Techniques to Positively Impact Runtimes.

Ian Sanderson, IQVIA, Reading, UK

ABSTRACT

This paper will assess the program development process, in relation to handling of big datasets, and will provide a framework for applying a range of programming techniques to actively manage program runtimes. Furthermore, we will investigate how to identify areas of improvement of existing programs through utilisation of log review programs and/or the SCAPROC procedure. We discuss a range of tools readily available to the majority of SAS® users including some well-known, DROPs, KEEPs, WHEREs, IF-ELSEs and PRESORTEDs, as well as those less frequently used; THREADING, SAS Views and HASH object merges. We hope that by using techniques suggested in this paper it is possible to future proof programs and adapt our current mind-set so that managing datasets of any size in an efficient manner may become second nature.

INTRODUCTION

When asked by my non-programming colleagues what programming is, I like to use a simple analogy. Let's put two points on a map calling the first point (A) our data, or start point, and the second point (B) our outputs, or endpoint. Then the possible routes we can draw from A to B will be our program and therefore, the process of tracing the route is our programming. Sadly, often due to time constraints, a lot of programmers focus so heavily on the endpoint that they often forget about the route they wish to take. This can lead to a false economy where one is so focused on not spending too much time on programming that we invariably do not consider the impact of the program syntax on the runtimes, which can inadvertently lead to greater delays when running or rerunning programs both during development and over the course of the study.

Advancements in patient data capture techniques have led to more information than ever being available at our fingertips. These data, when efficiently and appropriately managed, have helped to successfully guide, develop and support new medical interventions aimed at improvement of global heath. As technology advances and load balancing platforms (i.e. SAS GRID) move to the forefront, it is still important to remind ourselves of some of the tools available to a SAS programmer to help handle this data effectively and future proof our code. This paper will discuss a range of efficient programming techniques, from simple blanket rules that can guide the approach to initial program development, to more specialised methods of manipulating bulky datasets. We will also discuss how you can identify areas for improving existing programs in order to actively target and reduce program runtimes.

In general, the methods discussed in this paper can be applied to datasets of all sizes, however for the purposes of this paper when discussing large data, we're usually referring to datasets more than 1-2 gigabytes. It is always good to first assess the size of the study data (or anticipated size) and then adopt the most efficient programming practices for that study, remembering that some techniques take a bit more coding time depending on experience and familiarity of the programmer with them. Additionally, some of the techniques offered may potentially increase the runtimes so it is important, when implementing these, to regularly monitor the programs efficiency throughout the development process.

GETTING STARTED (PRE-PROGRAMMING)

During the preliminary stages of program development, we often write and execute code detecting any minor errors within our syntax as we go. This can present a very real problem for programming with large data as it can often take a substantial amount of time to execute each step. Indeed, more of our time can be spent waiting on steps running than writing the program. If there is anything we can do to make the dataset smaller at this stage, then the potential impact on development time can be quite profound.

TRIMMING THE FAT

One of the first things we can look to do with large data is reduce the number of trailing blanks or 'dead space' on character variables. Given the current FDA requirements for SDTM/ADaM data submission you should see that any character variables have been set to the maximum length of the variable content. However, quite often we still see (particularly during the earlier stages of a study) SDTM/ADaM datasets having their characters set to an arbitrary length of 200. It is easy to think of blanks as nothing, however, each blank space takes up 1 byte (B) of information. So, let us say you have a dataset of 20,000 observations and you have created a new character variable with a value of "Y", "N" or "NA", but gave it a length of 200. The created variable has added 200 * 20,000B = 4000,000B or

4MB to the final size of the file. Realistically though, we're only using at most two of those bytes per observation, which should be more like an increase of 0.04MB. Taking a step back therefore, we can look at our anticipated input data and determine whether there are redundant spaces. After which, we can determine the minimum length required to capture the 'actual' data and then use the ALTER TABLE statement in conjunction with MODIFY in SQL procedure to reassign the length of this variable. If you are working from raw data, which is often extracted from a database to the maximum variable lengths, it is useful at a study level to perform this modification immediately after extraction prior to performing any other programming. This is one of the most effective ways of establishing baseline program efficacies at a study wide level.

```
***Limiting length of INDSN variable VAR to $2***;
proc sql noprint;
  alter table INDSN
  modify VAR char(2);
quit;
```

Snippet 1: ALTER TABLE statement

Once we've removed any blank data we can then start to move on to looking at what we need to be able to write our program, this is where returning to the initial analogy of planning a route becomes extremely important. We need to decide where it is we want to end up, what is our point B, from this information we can then establish what variables we will need to get there. Armed with a list of necessary variables we can make use of very simple, yet effective, DROP or KEEP statements to make sure we're working with just the bare minimum in our program. Quite often you'll have variables included in the starting dataset that you have no intention of using or, in some case, there may be both character and numeric versions of variables. The general rule for using these statements is, if you are only removing a handful use a DROP statement, or else if you just need to keep a handful the KEEP statement may be better. Whether you use DROP or KEEP, it is what is extremely important is where you place your statement. Ideally, these statements should always be used on the dataset as it is being read (in), rather than as it is written (out). This is because SAS reads the full content of a source dataset one observation at a time to construct the desired output dataset. So, if we tell SAS to omit one or more variables as it reads the data in it is saving time by not reading their content and reducing the overall read/write time. As we'll see later quite often when working with big data the speed of the program is governed by the rate at which SAS can read and the write temporary datasets.

To illustrate the differences with simply changing the relative position of KEEP or DROP statements on large data we performed a simple SORT, SORT and MERGE with incrementally larger datasets. Both sorted by the same key variables and merged together, keeping 10 (out of 52) variables from the large dataset and 4 (out of 22) variables from the smaller, in addition to the key variables themselves.

Table 1: Impact of position of KEEP statement	**Median Change in runtime (%), *n=10***		
SORT/MERGE Details	**Test A***	**Test B***	**Test C***
Baseline: KEEP statement on the final out dataset resulting from the MERGE	-	-	-
KEEP statement on the two datasets going into the MERGE	-3.4	-5.4	-5.6
KEEP statement on the two datasets going into each of the SORT procedures	-27.7	-28.8	-24.2

* Based on a smaller dataset of ~10,000 observation and larger datasets (A-C) of ~500,000, ~5,000,000 and ~10,000,000 observations, respectively

We can immediately see that the earlier in the program (in this case at the point that data are read into the sorts) we selectively reduce the data, the greater the impact on our time savings. Furthermore, this also helps to illustrate, in terms of runtime, that the SORT procedure is more susceptible to being adversely affected by surplus variables.

Another consideration to bear in mind when choosing which variables to remove is where we potentially reap the greatest rewards. Looking at paired variables (i.e. numeric with a character counterpart) we tend to keep numeric data as these seem easier to work with. However, SAS stores numeric values with a default length of 8 meaning additional unused spaces often get overlooked. Instead, where relevant, we should be looking to drop the longer of the two variables to maximise our potential efficiency returns.

SUBSET OF DATA

Another possibility to consider, before you start writing your program is whether it would be beneficial to make your datasets even smaller while you write and debug code. Indeed, you may have reduced the data to the minimum that is required, but as we've not necessarily excluded any observations at this time you may find that the dataset itself is still difficult to work with. Particularly as you may need to run and rerun small sections of code, which can inadvertently introduce quite a lot of waiting time into the overall development process.

In these instances, it is often useful to write the whole program on a subset of the data. This can be done in several ways, firstly you could just subset for a specific subject or parameter and program for that. The downside is this may not save you much time as there may be many parameter specific rules to consider. The second method, that is a little more robust, is to make use of the SURVEYSELECT procedure. This allows us to produce a well-distributed subset of the data with a user specified number of observations. All we simply need to do is specify the number of observations we intend to work with. The third option is to use the OBS= argument on our first data-step. It is imperative however, that the program comments the use of these subsets fully so that once the full program has been written these sub-setting can be removed easily, and the program run on all of the data.

Our aim in working with a subset is to make savings during the program debugging stage of development. Although, strictly speaking, this method will not directly affect the runtime of the final program. As we've seen previously, working with smaller data generally leads to quicker execution which, when running, and re-running code can have a noticeable effect on overall program development.

PROGRAM DEVELOPMENT (PUTTING PEN TO PAPER)

After determining that only the required data are being included at the start of our program we then need to move on to the program development. It is here, in order to maintain an efficient program, that extra care should be taken when choosing how we want to get from A to B. It can be all too easy to lose any initial efficiencies we gained by reducing the size of the data. Furthermore, the initial time-saving ideas considered before diving into our program should absolutely be maintained and reproduced at regular intervals throughout the data. The general rule underpinning all of this is to not pick up unwanted baggage as we continue our journey towards our endpoint.

ADDING DATA

As previously discussed the variable length (and therefore type) can potentially hinder our programs efficiency. So, when looking at adding data, or deriving new variables, we need to keep this in mind. A common addition to existing data is our use of 'flag' variables throughout the Program. Often, we may want to attach a flag to one or more observations to keep track of specific subsets of data throughout the process. These can be required flags (i.e. population flags) or temporary programming flags (i.e. flags to allow the programmer to easily subset/order observations). Usually, we as programmers will settle for a quick FLAG = 0, 1, 2…*n*. This has the bonus that it can be sorted easily and requires very little code. However, with the default size of numeric variables (8 bytes) we can find that this quickly adds to the overall size of the data. Instead, it is suggested where possible that we use single/double length character variables (e.g. FLAG = "1" - "*n*" or "01" - "*nn*"). This will allow us to retain the ability to sort by the new flag, but now we've reduced the additional data to 1-2 bytes per observation.

Secondly, we should try to limit the amount of identifying information we add to the program until the very end. For example, if we were working with the lab data and we have the LBTESTCD data available. We should consider either dropping, or refrain from merging on, the LBTEST variable (which holds considerably more information) until one of the last steps in the program. Similarly, there may be several subject level variables or items needed for the final dataset that do not need to be used in derivations throughout the program. These items, if possible, should be dropped early and then later, before outputting the final dataset, merged back on. This helps protect our program from carrying unnecessary baggage throughout and hampering its performance.

SMART LOGIC

As we continue to work through the program we need to always have in the back of our minds a fundamental understanding of what we're asking SAS to do at each individual data-step and procedure. Indeed, this is where we'll often unknowingly be adding inefficiencies to our data. A substantial proportion of the 'wasted' time in programs stems from SAS reading observations that are not being used directly. We saw our first saving for this by using the KEEP/DROP statements. However, in addition to these, appropriate and regular use of IF-THEN/ELSE statements can mean that once an observation satisfies the criteria SAS no longer reads or considers that observation for the later portions of the statements.

Similarly, effective use of the WHERE statement rather than IF to subset data means that we can avoid the time-consuming processes of SAS reading the observations it doesn't need to. Remember that as a rule the 'WHERE' is performed before of processing the data and 'IF' is performed after the processing data but before returning results. So, what this means is if the WHERE statement is not satisfied by an observation then SAS does not bother reading the record into the program data vector (PDV). The PDV is essentially a space in the memory where SAS builds a dataset from scratch one observation at a time. Once you understand this principle it becomes easy to see how asking SAS to only add the necessary observations speeds up the process.

SORTING THROUGH THE DATA

It is often true that the most time-consuming steps, when working with very large datasets, come from our SORT procedures. This is because SAS needs to read every single observation, then ordering them and then writing each observation. Furthermore, there is a common misconception that SAS is only looking at the variables specified in the BY statement, however, this is not the case. Indeed, if you had a dataset with 25 variables and you're only sorting by

3 of those variable SAS is still reading the other 22 despite not needing to use them. As we saw earlier this use of the KEEP option on the input dataset really can have a marked effect on the runtime of this procedure.

It is usually assumed that adding more lines of code or in fact more steps can lead to increased run time. While for smaller datasets this may often be true this is not always the case when working with very large data. It has been suggested (Kuang, 2011) that breaking up large data and running multiple sorts can in fact speed up the process. This can be done using the (FIRSTOBS= and OBS=) arguments on a proc sort. The down side of this is that it can be unclear when it is best to use this approach.

An alternative to manually initiating this 'multi-step' method is to allow SAS to make more efficient use of the CPU capacity to perform its own multi-threaded processing. This can be done by specifying the THREADS option in the procedure. SAS will know that you would like it to use a multi-threaded approach and will split the data up (as it sees fit) over the specified available CPU cores and run the code in parallel. We can go one step further and add "OPTIONS THREADS CPUCOUNT=ACTUAL" to our global options, rather than adding the option to the specific procedure. This global option allows SAS to determine if threading would be best to use for any given. When this is done the log prints a simple note (e.g. "*NOTE: SAS threaded sort was used.*"). Furthermore, the specification of "ACTUAL" for the CPUCOUNT means that SAS can assess how many cores are available without the user needing to know the specific processing power. Alternatively, if the user wanted to limit this, simply specify the number (e.g. "CPUCOUNT=2"). It is important when assessing the use of this method to focus on the benefits to overall runtime. Threading will often result in a corresponding increase in the CPU time, as you'd expect, but it is the overall real time of the run that will be reduced.

To give a comparison between a standard SORT, SORT and MERGE (as discussed earlier) here we can see the combined benefits of both good use of the KEEP statement and a multi-threaded sorting approach.

Table 2: Standard vs. Threaded SORTs	**Median change in runtime (%), *n=10***		
SORT/MERGE Details	**Test A***	**Test B***	**Test C***
Baseline: KEEP statement on the final out dataset resulting from the MERGE	-	-	-
KEEP statement on the two datasets going into each of the SORT procedures	-27.7	-28.8	-24.2
KEEP statement on the final out dataset resulting from the MERGE – with THREADS option (CPUCOUNT=ACTUAL)	-15.7	-11.5	-12.4
KEEP statement on the two datasets going into each of the SORT procedures – with THREADS option (CPUCOUNT=ACTUAL)	-37.5	-28.1	-27.6

* Based on a smaller dataset of ~10,000 observation and larger datasets (A-C) of ~500,000, ~5,000,000 and ~10,000,000 observations, respectively

Allowing SAS to use a threaded approach and then combining this with a smart use of the KEEP statement appears to provide a stable and scalable performance boost to our SORT procedure further with minimal programming updates. Additionally, it is worth noting that particularly with the smaller data this approach doesn't appear to adversely affect runtime of the programs. Indeed, though this may need further investigation with much smaller datasets, one could argue that this could be a simple and effective tool to roll out at the study-wide level.

SKIPPING STEPS ALTOGETHER

We've touched on the consideration that a substantial proportion program runtime is consumed with continuously reading and writing datasets. So, by that logic, if we were able to skip one or more of these read/write cycles then we'd expect to see an instant reduction in the runtime. This therefore seems a good focus for our next efficiency gains.

SKIPPING STEPS ALTOGETHER: OPTIONS

The first and most straight forward way we can do this is by making use of the PRESORTED option for our SORT procedures. Quite often in defensively written programs the coder may have added one or more sort procedures before a merge, despite at the time of writing the program the data being in the correct order. It is good practice to write in such a way that if modifications are made earlier in the program issues will not be encountered by later steps. However, when working with large datasets this can lead to inefficiencies in runtimes. The way around this is simply by adding the 'PRESORTED' option to the first line of the procedure. This tells SAS to quickly look at the dataset being sorted and determine if it is already in the correct order. If this is the case SAS will print a simple note to the log confirming the data to be in the correct order and that the sort procedure was skipped. If the data are not in the correct order, then SAS will print a note to indicate this and perform the sort procedure as normal.

It can be argued that if the sorting is needed then you run the risk of potentially adding to the processing time as SAS will need to check to see if the order is already correct. However, this additional resource requirement is

minimal and if just one sort was skipped when working with a large dataset the benefits to overall runtime would far outweigh this minor drawback.

SKIPPING STEPS ALTOGETHER: INDEXES

Often considered as the most efficient method of working with large data in SAS is to make use of the indexing. SAS indexes allow the program to selectively target a small subset within a large dataset in the order required without sorting. Indexes can be thought of as reference cards of the specified variables that contain both the ascending order of the variables values as well as the relative position of the observations. This means that SAS can effectively find these observations quickly and already knows the order of them.

The index itself is created earlier in the program, examples below, by specifying the key variables likely to be needed most often later in the program for WHERE or BY statement. There are two types of index a simple index, which contains just one variable, and a composite index, made up of two or more variables.

```
***Simple Index Syntax***;
proc datasets lib=work;
  modify large; index create param;
run;

***Simple Index Syntax (SQL)***;
proc sql;
  create index param on large;
run;
```

```
***Composite Index Syntax***;
proc datasets lib=work;
  modify large; index create catparm=(cat param);
run;

***Composite Index Syntax (SQL)***;
proc sql;
  create index catparm on large(cat param);
run;
```

Snippet 2: Creating an INDEX for Simple and Composite Variables

With indexes there are two general rules to bear in mind when deciding if these are right for your program. Firstly, the initial index itself takes processing time to set up and needs to be used multiple times throughout a program to have efficiency returns, the more you use an index the more benefit you'll see on the overall program efficiency. Secondly, to gain a benefit from an index the target subsets of data should ideally be less than 25% the size of the initial dataset. So, for example, a population flag may not be a good target, however, PARAMCD may well be an ideal candidate as it will mean you can selectively retrieve the observations for a given parameter (using a WHERE statement) very quickly.

SKIPPING STEPS ALTOGETHER: VIEWS

Another useful method for cutting down on unnecessary processing time is by implementing SAS VIEWs. A SAS VIEW can be thought of as the blueprint or a shell of the dataset. It does not contain the value level data, instead it contains metadata that SAS can use to create a dataset. These VIEWs can be created in either a Data-step or with the SQL procedure and work on the basis that they don't fully complete the step until they are next called. For example, each time SAS executes a full data-step there is usually time required to fully read the desired input dataset(s), perform whatever functions/derivations are required, then write to the output dataset. Using a SAS VIEW instead creates a smaller temporary file that contains all the metadata required to execute the data-step. This is not done until the VIEW file is next called thereby skipping a full read/write cycle. It can be particularly useful for setting/merging data together before performing an overall SORT and it is also possible to set one VIEW with another VIEW or datasets into a new VIEW.

A good question may be, "why we don't just use views all the time?". The truth is that some of the SAS procedures (specifically heavily statistical procedures) require multiple passes (or reads) of the data to execute so as a result the when it reads the initial VIEW it needs to create an intermediate file that it can reread after each pass, and, given that a VIEW takes more processing time (CPU), this can defeat the point of using the VIEW in the first place. A general rule is that if the reason for merging or setting datasets together is simply to feed these directly into a simple procedure (SORT, REPORT, MEANS, TABULATE, SUMMARY), or you're just looking to do some quick mapping, then consider using a VIEW.

In the example below, a VIEW is created (B) in a data-step by adding "/ VIEW = <name>" after the DATA statement. Note that the VIEW should take the same name as the dataset which it represents. As we can see, when compared with a standard data-step approach (A), almost no time resources were used in the creation of the VIEW, in this case saving us just shy of 30 seconds. One thing to bear in mind is, a VIEW cannot be overwritten by a dataset of the same name, nor can it overwrite a dataset. This means that when VIEW is applied to create a dataset this cannot have a name identical to that of the VIEW.

```
***Standard approach (non-VIEW)***;
A       data large;
          set indat.large4 (keep=keyvar trtgrps largevar1-largevar10);
          trtgrpC = "Treatment "||strip(put(trtgrps, best.));
         run;

NOTE: There were 10136407 observations read from the data set INDAT.LARGE4.
NOTE: The data set WORK.LARGE has 10136407 observations and 13 variables.
NOTE: Compressing data set WORK.LARGE decreased size by 33.53 percent.
Compressed is 11538 pages; un-compressed would require 17357 pages.
NOTE: DATA statement used (Total process time):
real time           24.72 seconds
cpu time            11.42 seconds


***Defining a VIEW***;
B        data large1 / view=large1;
          set indat.large4 (keep=keyvar trtgrps largevar1-largevar10);
          trtgrpC = "Treatment "||strip(put(trtgrps, best.));
         run;

NOTE: DATA STEP view saved on file WORK.LARGE1.
NOTE: A stored DATA STEP view cannot run under a different operating system.
NOTE: DATA statement used (Total process time):
real time           0.00 seconds
cpu time            0.01 seconds
```

Snippet 3: Standard Data-step vs. defining VIEW for simple setting and mapping

SKIPPING STEPS ALTOGETHER: HASH OBJECTS

The last, and by no means least, efficient programming method this paper will discuss is the often overlooked, yet immensely powerful, HASH object. We've identified that most problems with efficiently manipulating large dataset stem from sorting and merging the data. What if it was possible to create your combined dataset from its constituent parts without needing to creating an index or sort it first? This is what a HASH object can do for you.

In short, a HASH object can be thought of as a look-up table created and accessed within the internal memory that has a built-in index. As the object exists in the internal memory there is no need to read or write to a server or disk. This means SAS can access and populate the data in the object extremely quickly. The syntax for HASH objects can be somewhat confusing at first, however, a good example (Snell, 2006) is to remember it as 4 simple steps:

1. 'Create', telling SAS to begin construction on the object
2. 'Define', tell SAS the structure your object will take
3. 'Fill', selecting the values which SAS will use to populate the object
4. 'Access', the final retrieval of the desired product of items from the object.

The creation step begins with a simple DECLARE statement, this is where the user names the HASH object. From there we can define the variable lengths, the key variable(s) and any data variable(s) that will go into the object, think of this section part as what variable(s) you're trying to merge on and what your BY statement would be. Noting the object is to be the smaller of the two datasets being joined. Once created and defined, SAS has an empty box that looks like our smaller dataset. Now we can use an explicit loop (i.e. DO UNTIL), a SET statement and the **<HASH NAME>.ADD()** command to populate the empty box. The reason this needs to be an explicit loop (rather than the standard implicit loop SAS uses when it sequentially reads data) is because SAS needs to know when exactly it should stop reading the records as we're directly accessing them, so no given order is expected. Finally, we've added our data, so the in-memory object is ready. All that is left is to make use of the very fast direct access we have to it to be able to retrieve values. This is where our built-in index, in this case the specified key variables, can be targeted. We use a second SET statement on our larger dataset and another explicit loop. We first create all the new variables (the items being merged on) as null, a combination of the CALL MISSING routine and LENGTH statement is great for this. Then the **<HASH NAME>.FIND()** command is used to find values in our object, based on its index keys, and bring them over to the larger dataset. To finish the entire process, we can use the OUTPUT statement to either write all observations or, by using an IF-THEN statement, to write selected observations to the file, that is it, our HASH object is used, and we've created a merged dataset at the point which we're reading it in, all in one data-step!

```
data OUTDAT;
    ***CREATE (create space for hash object)***;
    declare hash h_small();

    ***DEFINE (define details of content of hash object)***;
    length newvar1 newvar2 $10. newvar3 newvar4 8.;
    rc = h_small.DefineKey("byvar1", "byvar2");
    rc = h_small.DefineData("newvar1", "newvar2", "newvar3", "newvar4");
    rc = h_small.DefineDone();

    ***FILL (loop over the INDAT_SMALL to populate the hash object)***;
    do until (eof_small);
      set INDAT_SMALL end=eof_small;
      rc = h_small.add();
    end;

    ***ACCESS (loop over INDAT_BIG and merge the hash object with it)***;
    do until (eof_large);
      set INDAT_BIG end=eof_large;

      ***Create vars that are merging on before merging***;
      length newvar1 newvar2 $10. newvar3 newvar4 8.;
      call missing(newvar1, newvar2, newvar3, newvar4);

      rc = h_small.find();
      output;
    end;
  run;
```

Snippet 4: HASH object merge

It is important to note that it is not possible to set in a VIEW when creating a HASH object. This is because SAS need the actual data rather than the specification of the data which would be supplied by a VIEW. However, smart use of KEEP or DROP statements can be used here with the SET statements. As you can see from the table below using a HASH object merge to replace a standard SORT, SORT and MERGE results in an and exceptional reduction in program execution time.

Table 3: Standard SORT and MERGE vs. HASH object merge	**Median change in runtime (%), *n=10***		
SORT/MERGE Details	**Test A***	**Test B***	**Test C***
Baseline: KEEP statement on the final dataset resulting from the MERGE	-	-	-
HASH object merge, KEEP statement on the final dataset	-43.2	-41.6	-39.7
HASH object merge, KEEP statements on the SET statements	-43.2	-38.3	-41.7
Macro HASH object merge, KEEP statements on the SET statements	-37.8	-31.9	-36.3

* Based on a smaller dataset of ~10,000 observation and larger datasets (A-C) of ~500,000, ~5,000,000 and ~10,000,000 observations, respectively

If you're worried about remembering the full syntax for a HASH object, then the solution is a simple one. It is quite possible to adapt the above code example into a macro that can be called where just the variables, key and two datasets are specified. This helps protect the programmer from added coding time. Even despite the additional programming steps inevitably needed by the macro we can see (Table 3) the benefit of using a macro for this method remains close to 30% across all levels.

EXISITING PROGRAM REVIEW (WHY IS IT TAKING SO LONG?!)

It can be quite frustrating when we inherit a program that runs slowly. Generally due to time constraints it is imperative that we can quickly identify areas where existing programs can be streamlined and selectively target these areas. Furthermore, it is imperative that we as programmers can identify these potential drawbacks in our programs and learn to avoid them going forward. Rather than just reading through the code line by line we can develop a number of tools or simple programs that can help extract this information for us.

RETRACING YOUR STEPS

Usually log files are overlooked as review tools. It is correct, they can allow us to quick confirm when a program was last run and if there are any ERRORS, WARNINGS, or NOTES that should be addressed to adhere to good

programming practices. But that isn't where their usefulness stops. The wealth of information provided by the log file can also be a goldmine for programmers who are looking to review their program for efficiencies.

When looking for areas to improve an existing program the first step is to read in the logfile, then armed with this we can begin to parse the multitude of notes, runtimes and lines of code to target inefficiencies. To start with we can look at global options, to check that the THREADS option is applied. Afterwards, we can establish the point at which we have first read in the data, a separate PROC CONTENTS on these datasets and comparison against all variables used through the program or output in the final dataset can give us what we need for a suitable KEEP statement list for the start of the program. We can also check these datasets to see that the length specified in the output of the PROC CONTENTS matches the maximum length of the data, if not we can run the datasets through a reduce macro (Zhong, 2012). Finally, we can selectively target the SORT procedures, both to check the PRESORTED option is in use throughout, and to establish any SORT steps that are just needed to prepare the dataset for a merge. It is here we can look to replace one or more blocks of SORT and MERGE with our simple HASH object macro.

ANALYZING PROGRAMS: SCAPROC

If we are looking to take an even deeper dive into the program we can, along with the standard log file, use the SCAPROC procedure (SAS code analyser) to produce a more detailed log file. As indicated this is a SAS procedure and therefore needs to be run with the program itself. There are two quick ways to do this, firstly we can create a simple macro program with a %INCLUDE statement bracketed by the syntax for creating a SCAPROC (below, A) or we could use the command prompt (below, B).

A: Macro call method

```
%macro code_analyser(inPROG=, outLOG=);
  ***Begin the SCAPROC and specify log details***;
  proc scaproc;
     record "&outLOG." ATTR OPENTIMES;
  run;

  %include "&inPROG.";

  ***Write our analysis log file***;
  proc scaproc;
     write;
  run;
%mend code_analyser;
```

B: Command line method

```
sas program.sas -initstmt "proc scaproc;  record 'analysisFILE.txt' ; run;"
```

Snippet 5: Examples of SCAPROC execution

As you see, in section A, there are additional options that can be added to RECORD statement. In this example, the ATTR (attributes) provides information in the output file regarding the variables in each dataset. The OPENTIMES, which may be more relevant for our purposes, gives a detailed breakdown of the datasets, like access time, size (in bytes) and location of the file. The additional log file we gain is much easier to parse than a standard logfile. Indeed, additional information is placed in handy blocks immediately preceding the step to which they relate. Each information line begins with "/* JOBSPLIT:" before describing the details of the item. Listed below are a few of the key elements along with details and example questions this information may help us to answer.

```
/* JOBSPLIT: DATASET INPUT SEQ WORK.TEST0.DATA */
```

This line shows us the dataset type (INPUT/OUTPUT), what is called and where it was stored as well as what type of read pass SAS performed on the data (SEQ/MULTI, sequential or multiple, respectively). *Could we use a VIEW?*

```
/* JOBSPLIT: OPENTIME WORK.TEST0.DATA DATE:11SEP2018:10:51:49.82
PHYS:<pathway>\test0.sas7bdat SIZE:131072 */
```

Here, with the OPENTIMES option on, we can see the physical pathway of the stored dataset, its relative size and when it was accessed. *Do we need to consider efficiency tools for a dataset of this size?*

```
/* JOBSPLIT: ATTR WORK.TEST0.DATA INPUT VARIABLE:i TYPE:NUMERIC LENGTH:8 LABEL:
FORMAT: INFORMAT: */
```

On this line, with the ATTR option on, the log returns (one variable at a time so be careful with this option) the individual attributes of the variables including any formatting. *Are there lots of character variables set to 200?*

```
/* JOBSPLIT: ELAPSED 5  */
/* JOBSPLIT: PROCNAME DATASTEP */
```

Finally, towards the end of the information blocks we will see some of the most valuable information. In the first line we have the elapsed time for a given procedure and the second is the procedure type. It is this key information that would allow a user with minimal effort to very quickly benchmark and produce a summary of all the time spent on a given procedure type, determine if any one of the separate procedures is taking longer than the rest and immediately help us to target individual steps in the program to consider applying efficient programming techniques. *Where are our biggest problems and how much of an impact are they having on runtimes?*

CONCLUSION

While program runtime inefficiencies can easily be missed when working with small data, they can have a detrimental impact on the overall performance when working with very larger data. Fundamentally, there are no 'magic bullets' to solve all the issues that may be slowing down a program. However, by approaching program writing in a logical manner and armed with a basic understanding of how SAS performs the desired operations, we can identify and selectively target key problem areas for programs and start to actively manage our control over the runtime.

REFERENCES

Kuang, S. (2011). Efficient Techniques and Tips in Handling Large Datasets. Retrieved from https://www.lexjansen.com/wuss/2011/coders/Papers_Kuang_J_75005.pdf

Snell, G. P. (2006). Think FAST! Use Memory Tables (Hashing) for Faster Merging. Retrieved from http://www2.sas.com/proceedings/sugi31/244-31.pdf

Zhong, W. (2012). Efficiently Trim Character Variable Lengths to Fit Data, Reduce Dataset Size. Retrieved from https://www.pharmasug.org/proceedings/2012/CC/PharmaSUG-2012-CC17.pdf

RECOMMENDED READING

Agnihotri, K. (2014). Need for Speed in Large Datasets – The Trio of SAS® INDICES, PROC SQL and WHERE CLAUSE is the Answer. Retrieved from https://www.pharmasug.org/proceedings/2014/CC/PharmaSUG-2014-CC16.pdf

Becker, M. (2017). How Did This Get Made? Retrieved from https://www.lexjansen.com/pharmasug-cn/2017/DM/PharmaSUG-China-2017-DM01.pdf

Dosani, F. (2007). Two heads are better than one – Getting the most out of multiprocessing. Retrieved from http://www2.sas.com/proceedings/forum2007/195-2007.pdf

First, S. (2012). The SAS® Log: A Wealth of Data and Job Flow Information. Retrieved from http://support.sas.com/resources/papers/proceedings12/237-2012.pdf

Jain, V. (2010). Working Efficiently with Large SAS® Datasets. Retrieved from https://www.lexjansen.com/phuse/2010/cc/CC03.pdf

Johnson, J. (2012). The Use and Abuse of the Program Data Vector. Retrieved from http://support.sas.com/resources/papers/proceedings12/255-2012.pdf

Mason, P. (2017). Automatically create diagrams showing the structure and performance of your SAS code. Retrieved from http://support.sas.com/resources/papers/proceedings17/1104-2017.pdf

Raithel, M. A. (2005). The Basics of Using SAS® Indexes. Retrieved from http://www2.sas.com/proceedings/sugi30/247-30.pdf

Ray, R. (2002). Save Time Today Using SAS Views. Retrieved from http://www2.sas.com/proceedings/sugi27/p019-27.pdf

SAS. (n.d.). Sample 34301: Parse SAS® logs to extract performance and timing information. Retrieved from SAS: http://support.sas.com/kb/34/301.html

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Ian Sanderson
IQVIA
500 Brook Drive
Reading / RG2 6UU
Work Phone: 0118 450 1267
Email: ian.sanderson@iqvia.com
Web: www.iqvia.com