Paper: 1058-2017

Going Green
With Your SAS® Applications

Michael A. Raithel, Westat

Abstract

This paper shows how you can reduce the computing footprint of your SAS applications without compromising your end-products. The paper presents the *15 axioms of going green with your SAS applications*. The axioms are proven, real-world techniques for reducing the computer resources used by your SAS programs. When you follow these axioms, your programs will run faster, use less network bandwidth, use fewer desktop or shared server computing resources, and create more compact SAS data sets.

Introduction

In today's competitive business environments, SAS programmers cannot afford to be wasteful when using their organizations' computing resources. IT budgets are stretched to the limit; it is difficult to obtain funding for new software products; organizations are extending the time between server upgrades; it is hard to get approval for additional disk arrays and memory; and people are being told to make do with the older model PC's on their desktops. At the same time, IT workloads are growing; SAS programmers are being asked to turn out more reports, tables, graphs, and data sets; more concurrent batch programs are running on Linux, UNIX and Mainframe servers; and clients want their deliverables *on time*. These are just some of the factors that are putting pressure on the existing computing resources of many organizations.

On a daily basis, only a finite amount of computing power exists within an organization. So, you need to write SAS programs that process data, perform analysis, and create deliverables within the means of your organization's current computing *ecosystem*. You need to make the best use of existing computing resources to turn out your end-products without over-utilizing those assets. On shared servers, such as Linux, UNIX and Mainframes, wasteful SAS programs—programs that use excessive CPU time, I/O's, wall-clock time, etc.—affect all other tasks running on those servers. On workstations, wasteful SAS programs chew up valuable wall-clock time and may slow down other desktop applications. They can also negatively affect everybody's response times when they drag unnecessary amounts of data across corporate networks.

This paper introduces the *15 axioms of going green with your SAS applications*. These axioms are links to proven SAS programming techniques that facilitate efficient SAS programs. Learn the axioms and the underlying techniques to write programs that are effective while not being wasteful.

Note that is neither practical nor doable to cover every facet, every nuance, and every possible contingency of how the SAS constructs and techniques in this paper can be used. Readers are advised to digest the main concepts being presented in each axiom and then to refer to SAS online documentation, SAS conference papers, and SAS Press books for a deeper dig on the various subjects.

The 15 Axioms of Going Green with SAS Applications

1. Only take what you need.

Very often, we only need a handful of variables to perform analysis on our data. Similarly, we often only need observations that meet specific criteria instead of all of the observations in a SAS data set. When working with large,

very large, or extremely large SAS data sets, it does not make sense to create temporary SAS data sets that hold *all* of the variables and *all* of the observations found in the source data *when we do not need them*. The larger the data source, the more wasteful it becomes to hold unneeded data in our subset SAS data sets in terms of storage and processing time. Consequently, we should proactively limit the number of variables to only those needed for our analysis and similarly limit the number of observations.

We can limit the number of variables by judicious use of the DROP and Keep statements. Here are examples:

```
/* Using the KEEP= option to limit a SAS data set's "width"*/
data Car_Analysis1;
set  sashelp.cars(keep=make model type origin);
run;

/* Using the DROP= option to limit a SAS data set's "width"*/
data Car_Analysis2;
set  sashelp.cars(drop=mpg_city mpg_highway weight wheelbase length);
run;
```

We can limit the number of observations by using the WHERE and IF statements. Here are examples:

```
/* Using the WHERE statement to limit a SAS data set's length */
data Furniture_Analysis1;
set  sashelp.prdsal3(where=(state in("British Columbia" "Ontario" "Quebec"
"Saskatchewan")));
run;

/* Using the IF statement to limit a SAS data set's length */
data Furniture_Analysis2;
set  sashelp.prdsal3;

format yield dollar.;

yield = actual - predict;

if yield > 0 and state in("British Columbia" "Ontario" "Quebec" "Saskatchewan");

run;
```

Thinking about this axiom in geometric terms: You should strive to create SAS data sets that are as *narrow* and as *short* as possible without compromising the information you need for subsequent analysis.

2. Make a single trip to the well.

Sometimes we need to make several SAS data sets that are subsets of a large SAS data set. In such cases, it is tempting to use a separate DATA step for each of the subsets that we intend to make. However, that is not necessary and wastes computer resources when the large source data set is read multiple times. The bigger the source SAS data set, the more resources are wasted with each pass that is made to create a subsequent subset SAS data set.

Use a single DATA step to input the large SAS data set *once* and create your multiple subset SAS data sets. Here is an example:

```
/* Creating three output data sets from a single input data set. */
data Cars_USA(keep=make model type origin Enginesize cylinders horsepower)
     Cars_Asia(drop=mpg_city mpg_highway weight wheelbase length)
     Cars_Europe
```

```
       ;
set sashelp.cars;

select(Origin);
      when ("USA") output Cars_USA;
      when ("Asia") output Cars_Asia;
      when ("Europe") output Cars_Europe;
      Other;
end;

run;
```

In the example above, a single pass of the sashelp.cars data set is made to create three subset data sets: Cars_USA, Cars_Asia, and Cars_Europe. This is far more efficient than reading that data set three times to produce each data set separately.

3. SAS loves SAS data sets best.

SAS knows and understands SAS data sets better than it knows and understands any other type of data set. Once data are in SAS, you have programmatic access to all of the information about variables' lengths, labels, formats, informats, placement in the data set, etc. You also have access to the number of observations, data set size, and other data-set-specific types of information. So, it behooves you to keep the data that you are going to be processing numerous times with SAS in SAS rather than constantly reading it in from non-SAS data sources such as flat files or Excel spreadsheets.

Here is an example of reading a flat file into SAS so that it can be processed again and again and again.

```
data prodlib.Cars_Data_From_CSV    ;
infile 'C:\TEMP\DumpCarsSASDataSet.csv' delimiter = ',' MISSOVER DSD lrecl=32727
firstobs=2 ;
informat Make $5.
         Model $30.
         Type $6.
         Origin $6.
         DriveTrain $5.
         MSRP $9.
         Invoice $9.
         EngineSize best32.
         Cylinders best32.
         Horsepower best32.
         MPG_City best32.
         MPG_Highway best32.
         Weight best32.
         Wheelbase best32.
         Length best32. ;
  format Make $5.
         Model $30.
         Type $6.
         Origin $6.
         DriveTrain $5.
         MSRP $9.
         Invoice $9.
         EngineSize best12.
         Cylinders best12.
         Horsepower best12.
         MPG_City best12.
         MPG_Highway best12.
```

```
          Weight best12.
          Wheelbase best12.
          Length best12. ;
input Make $
      Model $
      Type $
      Origin $
      DriveTrain $
      MSRP $
      Invoice $
      EngineSize
      Cylinders
      Horsepower
      MPG_City
      MPG_Highway
      Weight
      Wheelbase
      Length
;
run;
```

Executing the above DATA step results in permanent SAS data set Cars_Data_From_CSV being created in the SAS data library represented by the prodlib libref. Once the SAS data set is generated, we can analyze and process it with additional DATA steps and with SAS procedures. If we did not create the permanent SAS data set, then we would have to go through the overhead of generating a SAS data set each and every time we wanted to analyze the data in the original CSV file.

4. Avoid SORTs.

Sorting is expensive in terms of the resources and time it takes to reorder the data. The larger the SAS data set being sorted, the more resources and time are consumed in the process. So, it is good to avoid sorts whenever you can. One of the best strategies for doing this is to require that data sets being given to you are already sorted in a sort order that you specify. This can be effective when you are being provided with data that are extracts of tables in a relational database such as SQL Server or Oracle.

When you are extracting data from tables in a DBMS and need to have the resulting SAS data set in a particular sort order, determine if you are able to perform the sort within the DBMS using the SQLGENERATION=DBMS option. That way the DBMS does all of the work and passes you a fully sorted SAS data set. See **Axiom #12 – Let the DBMS do all the work** for additional information.

In situations where you extract data from a non-SAS source and know the data are already in a particular sort order, you can *assert* the sort to SAS. Here is an example of doing that when inputting a flat file:

```
data prodlib.Cars_Data_From_CSV(sortedby = Make Model);
infile 'C:\TEMP\DumpCarsSASDataSet.csv' delimiter = ',' MISSOVER DSD lrecl=32727
firstobs=2 ;
      informat Make $5.
               Model $30.
            ......
```

In the example, the SORTEDBY data set option specifies that the permanent SAS data set being created, Cars_Data_From_CSV, is sorted by Make and Model.

You can also assert a sort in situations where you have already created a SAS data set from a non-SAS source and know that the source data are already sorted in a specific order. You can do this by modifying the directory information of your SAS data set to specify the sort order. Here is an example:

```
/* Assert a sort for the Snacks SAS data set. */
proc datasets library=work nolist;

modify New_Snack_Analysis(sortedby = product date);

quit;
```

After the DATASETS procedure is run, subsequent PROC CONTENTS will report that the data set is sorted by PRODUCT and DATE. That documents the sort order for other programmers who may use the data set.

5. <u>Don't disturb your data for trivial tasks.</u>

Sometimes SAS data sets are created in haste and proper attention is not given to providing them with meaningful metadata. That is; the variables do not have proper variable names, labels, formats, and informats. Also, the SAS data set itself might not have a descriptive label. This type of situation sometimes happens in environments where there is a lot of development going on to produce a final SAS data set that is to be delivered to a client—either external or within an organization.

When this condition occurs, it is tempting to create the various metadata by re-creating the data set with DATA/SET statements and specifying the proper variable labels, names, formats, and informats—as well as the data set label—all within the DATA step. However, the larger the SAS data set involved, the more wasteful using a DATA step to update the metadata becomes.

It is far more efficient to use the DATASETS procedure to update SAS data set metadata. Here is an example of using the DATA step to update SAS data set metadata.

```
/* Modifying formats, informats, labels, and renaming a variable. */
data sgflib.snacks(label = "New Snack Analysis Data Set");
set  sgflib.snacks;

      format price dollar6.2
             date  worddate.
             ;
      informat date mmddyy10.;
      label    product    = "Snack Name"
               date       = "Sale Date"
               advertised = "Advertised Price"
               holiday    = "Holiday Sale"
               ;
      rename Holiday = Holiday_Sale;

run;
```

Though the program above *does* fix issues with the formats, informats, labels, and names for the variables in the SNACKS SAS data set, it is not very efficient to run. It is inefficient because it reads the entire SNACKS SAS data set and creates a new copy of it, simply to fix data set metadata. If SNACKS is a small data set, then not much I/O, CPU time, and wallclock time are consumed. However, if SNACKS is big, then a lot of computer resources are consumed for several simple metadata changes.

SAS stores all of the metadata for a particular SAS data set in the descriptor portion of the data set, which is commonly located in the first physical page of the SAS data set file. The DATASETS procedure can be used to update this information by only processing the data set's descriptor page. So, instead of reading the entire SAS data set, it only reads the first page, updates the format, informat, label, or variable name information, and writes the updated first page back to the SAS data set. Consequently, it is much more efficient to use PROC DATASETS to update such information.

You can use the DATASETS procedure to modify SAS data set metadata like this:

```
proc datasets library=sgflib;
modify snacks;
      format price dollar6.2
             date  worddate.
             ;
      informat date mmddyy10.;
      label product = "Snack Name"
            date    = "Sale Date"
            ;
      rename Holiday = Holiday_Sale;

modify snacks(label = "New Snack Analysis Data Set");

quit;
```

The first MODIFY statement of the DATASETS procedure specifies that the SNACKS data set will have some of its metadata modified. Thereafter the FORMAT, INFORMAT, LABEL, and RENAME statements are executed to modify the attributes of the PRICE, DATE, PRODUCT, and HOLIDAY variables, respectively. The second MODIFY statement specifies a label for the SNACKS SAS data set. These direct updates to the SNACKS data set's metadata using PROC DATASETS are far more efficient than re-creating the entire SAS data set would be.

6. SAS has the best memory.

Buffers are a special portion of your computer's memory that SAS uses to store the data your programs are accessing. Each buffer is large enough to hold one page of a SAS data set. Data are transferred between storage devices and the buffers in units called data set pages. It is only when a SAS data set page is physically located inside of a buffer that a program can actually process the observations stored in it.

Larger numbers of buffers allow more SAS data set pages to be read into and reside in memory at a time. This can reduce the total number of I/O's required to process the data set and usually results in reduced wallclock time. The tradeoff is that more buffers use more memory. However, if memory is not an issue on your computer, then allocating more buffers is a good performance strategy.

In this example, we specify the number of data set buffers (BUFNO=), the number of index buffers (IBUFNO=) and the number of catalog buffers (CBUFNO=) that should be allocated for each of those particular entities. This means that every SAS data set the program opens will have that particular number of buffers allocated to it; as will every index and every catalog. The author uses 10 as his rule of thumb for the number of each type of buffer. But, you should experiment with those numbers to see what works best in your own computing environments.

```
/* 1. Allocate buffers. */
options bufno=10 ibufno=10 cbufno=10;
```

The SASFILE statement allows you to load an entire SAS data set into memory and keep it there for the duration of your program's execution. This can be helpful in cases where a particular data set is used again and again and again in a

program. Using SASFILE, the data set is read once and loaded into memory, expending whatever amount of I/O's and other resources are needed to do so. Subsequent accesses of that data set do not spend very much computer resources since the data are being read directly from memory. Here is an example of using the SASFILE statement:

```
/* 2. Use the SASFILE statement to load a data set into memory. */

/* Create data set to be used. */
proc sort data=sashelp.cars out=work.cars;
      by make type model;
run;

/* Load the data set into memory. */
sasfile work.cars.data open;

data Car_Analysis1;
set  work.cars(keep=make model type origin);
run;


data Car_Analysis2;
set  work.cars(drop=mpg_city mpg_highway weight wheelbase length);
run;


proc summary nway data=work.cars;
      class make type type;
output out=sum_make_type_model sum=;
vars msrp invoice;
run;

/* Close the in-memory data set and free the memory. */
sasfile work.cars.data close;
```

In this example, we first create a SAS data set, work.cars, via the SORT procedure. Next, we load work.cars into memory using the SASFILE statement with the OPEN option. The subsequent two DATA steps and the SUMMARY procedure all use the copy of work.cars that resides in memory. We finally free up the memory holding the work.cars SAS data set using the SASFILE statement with the CLOSE option.

Another excellent way to exploit memory is to use SAS Hash Objects. Due to the complex nature of SAS Hash Objects and the limitations on the size of this technical paper, they will not be covered here. However, there are many good papers and a few good books on SAS Hash Objects that you can access to learn more. See the References section of this paper for some suggested publications.

7. Modify your data in place.

When you need to concatenate two SAS data sets it is often tempting to do so in a DATA step. That is done by having the large SAS data set specified in the DATA statement and also first in the SET statement. The smaller SAS data set is then specified second in the SET statement. While this certainly works, it wastes computer resources because it is not really necessary for SAS to recreate the larger SAS data set.

A better strategy is to update the large SAS data set in place by concatenating the smaller SAS data set to it. This avoids SAS reading and moving all of the data set pages of the large data set into a new data set. SAS only has to copy all of the data set pages of the smaller data set and "paste" them to the "end" of the large data set. You can do this using the APPEND procedure. Here is an example:

```
/* Append smaller data set to main (base) data set. */
proc append base=New_Snack_Analysis
            data=Snacktran;

run;
```

In the example, the smaller SAS data set, Snacktran, is being appended to the larger SAS data set, New_Snack_Analysis.

When you append a data set to another, there are some restrictions on how the characteristics of the variables in the DATA= data set will be translated when the observations are appended to the BASE= data set. For example, variables that are exclusively in the DATA= data set will not be appended. There are similar restrictions on variable lengths, labels, formats, etc. found in the DATA= data set. Read the APPEND procedure documentation on *support.sas.com* for a thorough discussion of the restrictions.

Another strategy for modifying a data set in place is to use the MODIFY statement to update the observations in a *master* SAS data set to values found in a *transaction* data set. Here is an example:

```
/* Update master data set with transaction data set. */
data New_Snack_Analysis;
modify New_Snack_Analysis Snacktran;
      by product;

price = New_Price;
date  = today();

if _iorc_ = 0 then replace;

run;
```

In this example, each observation of Snacktran is read in and matched against the observations in New_Snack_Analysis by the value of PRODUCT. When there is a match—as denoted by _IORC_ = 0—then the values of PRICE and DATE are updated. Note that only the ***first*** observation in New_Snack_Analysis that is matched will be updated; subsequent observations will not be updated. So, this particular example is only effective when both data sets have unique values for PRODUCT.

This second example of the MODIFY statement uses a SAS index that was previously built from the patientid variable to directly read observations from the prodlib.hospital_file master SAS data set.

```
data prodlib.hospital_file;
set  prodlib.hosp_tran;
modify  prodlib.hospital_file key=patientid;

select (_iorc_);
      when(%sysrc(_sok)) do;      /* A match was found - update master */
            diagnosis  = newdiagnosis;
            replace;
      end;
      when (%sysrc(_dsenom)) do; /* No match was found, add trans obs */
            diagnosis  = newdiagnosis;
            output;
            _error_ = 0;
      end;
      otherwise;
end;
```

```
run;
```

In the example, each observation from the transaction file prodlib.hosp_tran is read sequentially and its value of patientid is matched against the index for the prodlib.hospital_file master SAS data set. When there is a match on patientid, the value of diagnosis is modified and the master SAS data set observation is replaced. If there is not a match, a new observation is created and appended to the master SAS data set. The %sysrc function is used to determine whether the attempted index search on a particular value of patientid was successful or not based on the value returned by the _IORC_ automatic variable. (See the SAS documentation on *support.sas.com* for more information about _IORC_ and %sysrc).

Though the MODIFY statement is very powerful and can save you a lot of computer resources when you update a master SAS data set via a transaction data set it can be tricky and you really have to know what you are doing. Duplicate observations in either or both data sets can result in you not getting the updates that you want. So, you should read the MODIFY statement documentation on *support.sas.com* to get a better feel for how to best use it to get the results that you want.

8. Right size your variables.

When you are creating very large SAS data sets with hundreds of thousands or millions of observations that have very long observation lengths, then looking at the size of the individual variables may be in order. You want to make sure variables are long enough to hold the information at hand, but not so long as to take up unneeded space.

Some easy variables to target for size reduction are:

- numeric variables holding categorical information
- character variables that were created via SAS functions where there was not an associated LENGTH statement

Numeric variables default to 8 bytes. So, if you are simply storing values of 1, 2, 3, 4 to specify *1 = USA, 2 = Europe, 3 = Asia, 4 = Other*, then you are wasting many bytes per observation. Note that the smallest allowable number of bytes for numeric SAS variables is 3-bytes on all operating systems; except for z/OS where it is 2-bytes. So, in cases where you are storing values of 1, 2, 3, 4, you can save 5 bytes if you were to use the smallest possible value of 3-bytes on all platforms except z/OS. On z/OS, you can save 6-bytes.

Character variables created by SAS functions default to a size of 200 bytes. So, if you are storing character strings much less than 200 bytes in them, then you are potentially wasting a lot of space per observation by accepting the default size.

When you do have very large SAS data sets, consider auditing the variables to see which ones can be reduced in size without compromising the data that they hold. Reducing variable sizes not only results in smaller SAS data sets that take up less storage space, but faster processing of those data sets because fewer I/O's are needed to read them.

Here is an example of a before and after:

```
/* Variables with excessive lengths. */
data cars;
set  sashelp.cars;

keep region tot_cylinders tot_weight newmodel;

select (origin);
      when("USA")    region = 1;
```

```
      when("Europe") region = 2;
      when("Asia")   region = 3;
      other          region = 4;
end;

tot_cylinders = cylinders;
tot_weight    = weight;

newmodel = tranwrd(model,"dr", "drive");

run;

/* Right-sized variable lengths. */

data cars2;
set  sashelp.cars;

length region $1.
       tot_cylinders 3.
       tot_weight 3.
       newmodel $45;

keep region Tot_cylinders Tot_weight newmodel;

select (origin);
      when("USA")    region = "1";
      when("Europe") region = "2";
      when("Asia")   region = "3";
      other          region = "4";
end;

Tot_cylinders = cylinders;
Tot_weight    = weight;

newmodel = tranwrd(model,"dr", "drive");

run;
```

In the first DATA step, variables region, Tot_cylinders, Tot_weight, and newmodel have lengths of 8, 8, 8, and 200, respectively for an observation length of 224 bytes. However, the largest values stored in those variables have lengths of 1, 3, 3, and 45, respectively, for a total of 52 used bytes. (*Remember that 3-bytes is the smallest allowable length of a numeric variable in SAS for Windows*). So, there are an extra 172 bytes of unused storage per observation. The second DATA step fixes the variable size issues by judiciously specifying the length of all variables that will be written to the cars2 SAS data set. This results in a SAS data set that takes up much less space but still holds the necessary information.

In addition to the examples above, you can also consider examining large character variables to determine the exact size of the largest character string stored in them. Then, you could resize those variables so that they are not longer than they need to be.

9. Only do what you need to.

When you are writing your SAS programs, think about how you can minimize the total number of DATA and PROC steps in your code. Each pass of SAS data sets consumes computer resources and wallclock time. So, it is more efficient to limit your program to the smallest number of DATA and PROC steps that get the job done. No more; no less. Here is an example:

```
/* 1. Too Many Steps. */

data work.cars;
set  sashelp.cars(where=(origin="USA"));
run;


proc sort data=work.cars out=work.cars;
      by make type model;
run;


proc summary nway data=work.cars;
      by make type type;
output out=sum_make_type_model1 sum=;
vars msrp invoice;
run;


/* 2. Only doing what you need to do. */
proc summary nway data=sashelp.cars(where=(origin="USA"));
      class make type type;
output out=sum_make_type_model1 sum=;
vars msrp invoice;
run;
```

In the first block of the example, a DATA step creates a subset of the sashelp.cars SAS data set. That subset data set is sorted into the specified order and then fed into the SUMMARY procedure. The resulting SAS data set, sum_make_type_model1, is the objective of the program.

The DATA step and SORT step can be eliminated by effectively performing the subsetting and "sorting" via facilities of the SUMMARY procedure. That is exactly what is done in the second block of this example.

The point is that you should look for opportunities like this to simplify your SAS programs so that they use the very minimum number of passes at the data to get the results that you want.

10. Clean up your room.

When processing very large SAS data sets where you are writing a lot of data to the WORK library, you can soak up a lot of space on your hard-drive or on a shared server disk drive. This may be an issue if you have a constrained workspace environment. You can address this issue by deleting intermediate SAS data sets from your WORK directory at the points where they are no longer needed in your SAS programs.

You can choose to specify which data sets are to be deleted, choose which ones are not to be deleted, or have all data sets in the entire library deleted. Here is an example of each:

```
/* Delete SAS data sets by listing those to be deleted. */
proc datasets library=work nolist;

delete cars cars2;

quit;
```

```
/* Delete SAS data sets by listing those to be kept. */
proc datasets library=work nolist;

save cars2;

quit;


/* Delete SAS data sets - entire directory. */
proc datasets library=work kill nolist;

quit;
```

Discriminating use of the DATASETS procedure to delete SAS data sets that are no longer needed can help your SAS programs to live within the means of the temporary disk storage available to them.

11. Leave a small footprint.

SAS data set compression not only reduces the size of SAS data sets, but it also promotes faster performance of programs that process those data sets. So, it is a good idea to compress very large SAS data sets. There are two types of compression: Run Length Encoding (COMPRESS=YES) and Ross Data Compression (COMPRESS=BINARY). The former is best to use when your observations have mainly character variables; the latter, for when you have mainly numeric variables.

You can compress individual SAS data sets via the COMPRESS= data set option or all data sets via the COMPRESS systems option. Here are some examples:

```
/* Compress a SAS data set when it is created.  Run Length Encoding compression.*/
data prdsal2(compress=yes);
set  sashelp.prdsal2;
run;

/* Compress a SAS data set when it is created.  Ross Data Compression.*/
data prdsal2B(compress=binary);
set  sashelp.prdsal2;
run;

/* Enable compression for all SAS data sets. */
options compress=yes;

data prdsal2C;
set  sashelp.prdsal2;
run;
```

The first example uses Run Length Encoding compression, while the second data set is compressed using Ross Data compression. In the third example, all SAS data sets created after the execution of the OPTIONS statement will be compressed using Run Length Encoding compression.

12. Let the DBMS do all of the work.

SAS has worked with some major database management system (DBMS) vendors to enable “in-database procedures”. In-database procedures permit the native DBMS to perform particular SAS procedures within the database itself. In-database procedures can reduce the elapsed time of SAS programs and reduce network bandwidth when large amounts

of data are processed within the database by its native engines and smaller result sets are served back to the executing SAS programs.

SAS has instrumented several procedures, including PROC FREQ, PROC SUMMARY, and PROC TABULATE to perform in-database processing in DBMS's such as Oracle and Sybase. You can direct SAS to use in-database processing via the SQLGENERATION=DBMS option. Here is an example:

```
libname oracledb ORACLE user='ProdUser' orapw='PROD$427' path='Oracledb'
schema='ledzep';

options sqlgeneration = dbms;

proc options option=sqlgeneration;
run;

proc summary data=oracledb.tblBIGDATA ;
      class STATE_CODE;
      var   Tot_Revenue Tot_Expenditures Tot_TaxesPaid Tot_Profit;
      output out=work.summ1 sum=;
run;
```

Above, the LIBNAME statement allocates a production Oracle data base. The OPTIONS statement, SQLGENERATION=DBMS, specifies for SAS to perform in-database processing of SAS procedures whenever possible. The OPTIONS procedure dumps the current value of SQLGENERATION to the SAS log to document that SQLGENERATION is enabled. The SUMMARY procedure accesses the tblBIGDATA Oracle table, summarizing specific variables and storing the result set in the summ1 SAS data set in the Work SAS data library. Oracle performs the summarization within the database and passes the summ1 data set to SAS. The large DBMS table stays put in the Oracle database, and we get a small summarized SAS data set.

Note that you must license the SAS/Access product that corresponds to your DBMS, such as SAS/Access Interface to Oracle or SAS/Access Interface to Sybase, in order to be able to use in-database procedures. Also, check the documentation on *support.sas.com* to determine in-database processing is available for your particular DBMS.

13. Have your own point of view.

An alternative to building and storing a SAS data set is to create a SAS data set View. A *view* stores the information about how a data set should be built in a small descriptor file whose extension is .sas7bvew. Once a view is created, it can be used as the input *file* to a DATA step or a PROC step. When a view is input to a SAS step, SAS reads in the various files, performs the various calculations, and writes out the various files that were originally programmed into it. So, the view acts a lot like a stored SAS program. Here is an example:

```
data cars2/view=cars2;
set  sashelp.cars;

length region $1.
       tot_cylinders 3.
       tot_weight 3.
       newmodel $45;

keep region Tot_cylinders Tot_weight newmodel;

select (origin);
      when("USA")    region = "1";
      when("Europe") region = "2";
```

```
      when("Asia")    region = "3";
      other           region = "4";
end;

Tot_cylinders = cylinders;
Tot_weight    = weight;

newmodel = tranwrd(model,"dr", "drive");

run;


proc summary nway data=cars2;
      class region;
      var tot_weight;
output out=sum_weights sum=;
run;
```

In this example, we create a view named cars2. That view is programmed to input the sashelp.cars data set, perform some data manipulations, and output observations containing only four variables: region, Tot_cylinders, Tot_weight, and newmodel. When the data step is executed, a file named cars.sas7bvew is created in the SAS work library. The view file only contains the instructions you see in the data step, above.

When the SUMMARY procedure executes, it reads the cars2 view file and executes the instructions found there. As previously stated, those instructions are to input the sashelp.cars data set, perform some data manipulations, and output observations containing four variables: region, Tot_cylinders, Tot_weight, and newmodel. Those observations are fed directly into the SUMMARY procedure.

There are other benefits to using views besides saving disk space. SAS always uses a current version of the data sets input to the view, so data surfaced by the view is dynamic, but a permanent SAS data set would need to be recreated whenever any of its input data sets change. You can limit access to sensitive data fields in the original SAS data sets by dropping them in the views that you create for them. If you have SAS/Connect software, you can join tables from different computing platforms in a view.

14. Do it some other time.

It is no secret that in most organizations the servers and the networks are the busiest during the main work hours of 9:00am to 5:00pm. If it makes sense, business-wise, consider running your SAS programs in batch during the less-busy hours. Doing so will take some of the burden off of the heavily used computing resources and will often lead to faster turn-around times for your own programs.

You run SAS programs in *batch* by specifying where your operating system can find the program, when to run the program, where to store the log and list files, and sometimes which other programs should be executed next. Operating systems have varying ways for how SAS programs are scheduled to execute in batch. Let's take a look at how this is done in the Windows workstation environment.

On Windows, you create a *.bat* file and use the *Windows Task Scheduler*. The .bat file is simply a text file wherein you put the full path of the SAS executable, the path to the SAS config file, the name of the program to execute, destinations for the log and list files, and other pertinent SAS options that can be specified at SAS invocation time. For example, this .bat file:

```
C:\Big Project\Production Programs\Midnight_Special.bat
```

…could contain the following instructions:

```
start C:\Progra~1\SAS94\x86\SASFoundation\9.4\sas.exe
      -icon -noterminal -nosplash -noxwait -noxsync
      -CONFIG C:\Progra~1\SAS94\x86\SASFoundation\9.4\SASV9.CFG
      -SYSIN "Q:\SAS_code\midnight_special.sas"
      -LOG   "Q:\SAS_code\midnight_special.log"
```

The instructions specify that when the .bat file is submitted to the operating system, SAS is to execute the *midnight_special.sas* program in the *Q:\SAS_code* directory and to store the log in that same directory. The .bat file specifies a number of SAS options as well as the location of the SAS config file.

You access the Task Scheduler on Windows by clicking on the following:

Start ➧ All Programs ➧ Accessories ➧ System Tools ➧ Task Scheduler

Once you open the Task Scheduler window, follow the prompts to schedule your SAS program for execution.

Batch programs can be scheduled via the crontab facility in UNIX and Linux environments. Mainframe computers often have CA-7 scheduling software installed for scheduling SAS (and other) programs for execution.

15. Let the BIG iron do the heavy lifting.

Many organizations have powerful computers such as mainframes, Linux servers, UNIX servers, or Windows servers to facilitate processing of large data sets and databases. Programmers with access to these machines can utilize the faster I/O transfer rates, increased memory, faster processor speed, and larger temporary disk work areas. If your organization has such a server and SAS is installed on it, then consider storing your large SAS data sets on the server and running your SAS programs that process those data sets on the server. This scenario works well when you have SAS/Connect software installed on both your PC and the server. SAS/Connect allows your Windows SAS programs to communicate with SAS on the server.

Here is an example of submitting a SAS program to a Linux server from a PC.

```
/*********************************************************************/
/* Inform SAS/CONNECT that we are using TCP/IP to connect to sas427 */
/*********************************************************************/
options comamid=TCP remote=sas427;

/*************************************************/
/* Signon to sas427 using a LINUX script         */
/* User is prompted for LINUX login and password */
/*************************************************/
signon 'C:\Program Files\SAS94\x86\SASFoundation\9.4\connect\saslink\tcpunix.scr';

/*************************************************************/
/* RSUBMIT this block of code to run on the sas427 server */
/*************************************************************/
RSUBMIT;

/*************************************************************/
/* Allocate the LINUX directory that contains SAS data sets */
/*************************************************************/
libname LINUXLIB "/home/marsyst/sasuser";
```

```
  /*************************************************/
  /* Summarize data on the Linux Server            */
  /*************************************************/
  proc summary nway data=LINUXLIB.demo05;
       class gender d_race;
       var c_days f_days;
  output out=LINUXLIB.Gender_Race_Sum sum=;
  run;

  /*******************************/
  /* Print out the results.      */
  /*******************************/
proc print data=LINUXLIB.summ1;
run;

/*****************************************************/
/* End of block of code submitted to the sas427 server */
/*****************************************************/
ENDRSUBMIT;
```

First communication is established between the PC and the Linux server via the information in the OPTIONS statement and the execution of the SIGNON statement. Then, all of the code between the RSUBMIT and ENDRSUBMIT is executed on the Linux server. That means that the millions of observations in the LINUXLIB.DEM05 SAS data set are summarized using the powerful processors, memory, and disk arrays on the Linux server. SAS log and list files are surfaced back on the PC. Consequently, all of the work is done on the BIG iron.

Conclusion

This paper introduced the *15 axioms of going green with your SAS applications*:

1. Only take what you need.
2. Make a single trip to the well.
3. SAS loves SAS data sets best.
4. Avoid SORTs.
5. Don't disturb your data for trivial tasks.
6. SAS has the best memory.
7. Modify your data in place.
8. Right size your variables.
9. Only do what you need to.
10. Clean up your room.
11. Leave a small footprint.
12. Let the DBMS do all of the work.
13. Have your own point of view.
14. Do it some other time.
15. Lst the BIG iron do the heavy lifting.

These axioms are prompts to proven SAS programming techniques that facilitate efficient SAS programs. Make these axioms and their underlying techniques an integral part of your SAS programming repertoire and enjoy having the most efficient and computer resource usage green SAS applications in your organization.

Disclaimer

The contents of this paper are the work of the author and do not necessarily represent the opinions, recommendations, or practices of Westat.

References

Raithel, Michael, and Rhoads, Mike. 2009. "You May Be A SAS Energy Hog If…" SAS Institute Inc. 2009. Proceedings of the SAS® Global Forum 2009 Conference. Cary, NC: SAS Institute Inc.
Available: http://support.sas.com/resources/papers/proceedings09/041-2009.pdf

Raithel, Michael A. 2014. "Did You Know That? Essential Hacks for Clever SAS Programmers: Over 100 Essential Hacks to Make Your Programs Leaner, Cleaner, and More Competitive." Bethesda, Maryland: Michael A. Raithel
Available: http://tinyurl.com/zfd67rh

Raithel, Michael, A. 2007. "Process Your SAS data Sets Anyway, Anyhow, Anywhere You Choose With SAS/Connect" SAS Institute 2007
Proceedings of the SAS Global Forum 2007 Conference. Cary, NC: SAS Institute Inc.
Available: http://www2.sas.com/proceedings/forum2007/241-2007.pdf

Dorfman, Paul M.; Vyverman, Koen. 2009. "The SAS® Hash Object in Action." SAS Global Forum 153-2009, Washington DC.
Available: http://support.sas.com/resources/papers/proceedings09/153-2009.pdf

Burlew, Michele M. 2012. SAS hash Object Programming Made Easy. Cary, NC: SAS Institute Inc.
Available: http://tinyurl.com/hqlt6so

SAS online documentation.
Copyright © SAS Institute Inc., SAS Campus Drive, Cary, North Carolina 27513, USA. All rights reserved
Available: http://support.sas.com/documentation/index.html

ACKNOWLEDGMENTS

The author would like to thank Westat management for supporting his participation in SAS Global Forum 2017.

The author would also like to thank retired Westat Vice President Mike Rhoads with whom he collaborated on the 2009 paper "You May Be A SAS Energy Hog If…" Many of the techniques in this paper were previously published in that one; which goes to show you that solid programming efficiency techniques never go out of style.

Thanks are also due to Westat Senior Systems Analyst Stan Legum for reviewing a draft of this paper and offering sage suggestions for its improvement.

CONTACT INFORMATION

If you have any questions about this paper or would like to provide feedback, you can contact me at the following email address: michaelraithel@westat.com