Paper 041-2009

You Might Be a SAS® Energy Hog If ...

Michael Raithel and Mike Rhoads, Westat, Rockville, MD

ABSTRACT

With heightened concern about global warming and rapidly-increasing fossil fuel prices, improving the energy efficiency of our home heating systems, appliances, and automobiles has become a priority for many of us. "Energy hogs" who waste energy, whether knowingly or inadvertently, contribute in a major way to the world climate and energy crises. Conversely, by modifying some of our behaviors to use energy more efficiently, we become more responsible citizens of the world, and can actually save some of our own money by consuming less energy.

You can apply similar principles to your SAS programming. While it's true that revising your SAS code will probably not save any cuddly animal species from extinction, employing more energy-efficient SAS programming techniques will make you a better corporate citizen by reducing the load on your enterprise's servers, disk storage, and network bandwidth. And, for a more direct payoff, your own programs may run more quickly, letting you impress the boss with your speedy turnaround.

This paper identifies some major clues that you may be a SAS Energy Hog and describes alternate techniques that will make your programs more efficient and friendlier to your corporate computing ecosystem.

INTRODUCTION

Though it is light-hearted in tone, the goal of this paper is to ask you, the reader, to take a serious look at the corporate computing resources that your SAS programs consume. All computing infrastructures have a finite amount of computing resources available to them. The amount of available processor time, memory, disk controller I/O, network bandwidth, and disk space are bounded by the equipment your organization has purchased and the configuration that your organization has established. When the demand for computing resources falls within the boundaries of what can be handled within your organization, work gets done in a timely fashion. But, when the demand for computing resources exceeds the capabilities of the available IT infrastructure, work is delayed, resulting in programmers waiting longer for their programs to complete and end-users waiting longer for the information that they need.

SAS Energy Hogs write programs that do not make effective use of computing resources. They are usually so focused on writing code that produces their deliverables that they inadvertently litter their programs with energy-wasting constructs. As more and more SAS programmers—and other programmers too—in an organization submit inefficient programs to the corporate computing infrastructure, the additional overhead of the inefficiencies that bloat these programs puts an unnecessary strain on corporate systems. If the superfluous overhead of each program were trimmed off by the judicious use of efficient programming techniques, then corporate computers could process more programs in a given time period, leading to faster turnaround times and a greater volume of completed programs. But how do you know if you are being a SAS Energy Hog, and what are some of the programming efficiencies that you can weave into your SAS programs?

This paper highlights many of the most common areas where SAS programmers typically waste computing resources. It provides examples of inefficient SAS programs and the corresponding energy-efficient SAS code. The paper explains the recommended programming constructs and why they are more efficient. It also provides references to additional information about the more complex energy-saving programming techniques. After reading this paper, you should be able to use the information that it presents to write more energy efficient SAS programs and make sure that you are not being a SAS Energy Hog.

YOU MIGHT BE A SAS ENERGY HOG IF ... YOU CARRY AROUND UNNECESSARY WEIGHT

One of the many factors that affects gas mileage is the amount of weight that you carry around in your vehicle. If you have been carrying 200 pounds of books in your trunk for the last 6 months because you keep forgetting to donate them to your local library, you are not being as energy-efficient as you might be.

GET RID OF UNNECESSARY VARIABLES AT THE EARLIEST POSSIBLE OPPORTUNITY

In SAS, you can waste energy and processing resources by carrying around unnecessary variables. One way you can do this is by creating array indexes and other temporary variables in a DATA step, and then failing to delete them from the data set(s) you are creating in the step. These unnecessary variables not only take up space and waste storage resources, but they make your program harder to understand, and in some instances their presence can cause hard-to-detect problems in a subsequent step that uses the data set as input.

KEEP and DROP are the methods most often used to restrict the number of variables. You can either DROP variables that you know are unnecessary, or KEEP only the variables that you need. Consider the following program:

```
data mylib.KeyAnalyticVariables;
set  mylib.DemographicData;
keep id sex educ age income AgeRecode;
/* Calculate AgeRecode using age */
run;
```

This is a good start, since you are using a KEEP statement to limit the number of variables. Note, however, that the KEEP statement affects the program's *output*, not its *input*. (DROP works the same way.) If you started out with 250 variables in your original data set, all of these variables will be brought into the Program Data Vector for the DATA step, although the ones not mentioned in the KEEP will not appear in the output data set. Therefore, you can be even more efficient by specifying the KEEP= or DROP= data set option for your input data set, so that variables that are not needed for processing are not brought into the step at all.

```
data mylib.KeyAnalyticVariables;
set  mylib.DemographicData (keep = id sex educ age income);
/* Calculate AgeRecode using age */
run;
```

GET RID OF UNNECESSARY OBSERVATIONS AT THE EARLIEST POSSIBLE OPPORTUNITY

You can also use resources unnecessarily by carrying around extra observations. Perhaps you want to create a series of income variables for the males in your data. You could get correct output with the following program:

```
data mylib.IncomeVarsForMales;
set  mylib.DemographicData
  (keep = id sex /* vars needed as basis for new variables */);
/* Create 10 new income variables */
if sex = 'M';
run;
```

Here you are being somewhat energy-efficient by using a subsetting-IF statement to keep only the observations that are male in the data set that you are creating. This will work correctly, but you are performing all of the variable creation for females as well as males, only to eliminate them at the end of your step with the subsetting-IF statement. You can improve the situation simply by placing the subsetting-IF statement earlier in your step, directly after the SET statement. Even better, use the WHERE= data set option so that observations that are not male are not brought into the DATA step at all:

```
data mylib.IncomeVarsForMales;
set  mylib.DemographicData
  (keep = id sex /* vars needed as basis for new variables */
   where = (sex = 'M'));
/* Create 10 new income variables */
run;
```

These examples suggest a useful general principle: to maximize your SAS energy efficiency, get rid of unnecessary variables and observations as early in your program as possible.

SQUEEZE OUT EVEN MORE WEIGHT WITH LENGTH STATEMENTS AND DATA SET COMPRESSION

Even when you have removed all of your unnecessary variables and records, there may still be some hidden weight that you can get rid of to maximize the efficiency of your programs. In order to take advantage of these techniques, we need to think more closely about how SAS stores data.

Let's assume that you have a data set with just a handful of variables, including a character variable for LastName. First, although it may be too obvious to state, make the length of the variable suit the information that it holds. Don't use a length of 200 for LastName if you know that you will never have a last name containing more than 30 characters, for instance. Instead, specify the maximum possible length in a LENGTH statement.

The real difficulty with text items such as names and addresses is their actual lengths vary greatly from record to record. Even though you may have a lot of relatively short last names (Jones, Lee, Lu, and Smith) in your database, you need to make the variable long enough to hold names such as Saltalamacchia and Rodgers-Cromartie. The problem becomes particularly acute when you have text fields (such as special notes or comments) that are completely empty for most of your observations but may have extremely long values when they are actually used.

This is an issue because SAS does not contain a native data type for varying-length character strings, as do most other programming languages and data storage architectures. Fortunately, SAS now offers data set compression to streamline the storage of data sets containing a lot of wasted space in character variables. Simply specify COMPRESS=CHAR as a data set option when creating your data set. This causes SAS to apply a compression algorithm that produces much smaller data sets in situations where you have multiple occurrences of repeating characters, such as trailing spaces at the end of character variables. For example:

```
data mylib.DataWithLongTextFields (COMPRESS=CHAR);
/* Other DATA step statements */
run;
```

You may also be able to squeeze extra weight out of numeric variables. SAS by default uses 8 bytes to store numeric variables, but as long as your variables only contain integer values you can typically use a LENGTH statement to store them in as few as 3 bytes (2 in a few operating environments) without any loss of precision. (It is almost never a good idea to store noninteger values using less than the full 8 bytes.) The online SAS documentation indicates the largest number that can be stored in a given number of bytes without losing precision: search for "numeric variable length precision") in the Companion manual for your operating system. The following example illustrates the use of the LENGTH statement:

```
data mylib.DataWithSmallIntegerValues;
infile ... ;
input ... ;
length Age NumberOfKids 3;
run;
```

You can also use data set compression to reduce the storage required by data sets with numeric variables, using the COMPRESS=BINARY data set option. You can find an excellent discussion of storage considerations for numeric data in Gorrell (2007). Using this technique, you might rewrite the previous example as follows:

```
data mylib.DataWithSmallIntegerValues (COMPRESS=BINARY);
infile ... ;
input ... ;
run;
```

Bear in mind that using either COMPRESS=CHAR or COMPRESS=BINARY requires SAS to use some extra processing resources to compress the data set when creating it and to uncompress it when you need to read it. In most cases this shouldn't be an issue for you–dealing with much smaller data sets reduces input/output operations and thus will normally make your jobs run faster, in addition to saving on disk storage. The extra processing resources may only be an issue if you are in a chargeback environment where the charging algorithm is primarily determined by CPU usage.

A final option to consider for numeric variables is whether they are truly "numeric" at all. If your values are numeric codes rather than actual quantities, you may want to store them as character rather than numeric, since character variables may be stored in as little as one byte.

YOU MIGHT BE A SAS ENERGY HOG IF ... YOU MAKE UNNECESSARY TRIPS

When you are driving, you consume a certain amount of gasoline every time you start a trip, no matter how short. One of the best ways to maximize the number of miles you get per gallon is to plan ahead, so that you can combine several errands into a single trip rather than making an entirely separate trip for each task.

DON'T PASS THROUGH YOUR DATA MORE OFTEN THAN NECESSARY

Energy efficiency in SAS works in much the same way. SAS code can be so compact and powerful that it's easy to forget that most DATA and PROC steps process all of the records in one or more data sets, even if the step takes up only a few lines in your program. The following is an extreme example of an energy-inefficient program:

```
/* Read data from external flat file */
data One;
infile Rawdata;
input id $char8.  (income1-income8) (9.);
run;

/* Compute total income */
data Two;
set  One;
TotIncome = sum(of income1-income8);
run;

/* Compute income recodes */
data Three;
set  Two;
/* Recode variable calculations go here */
run;
```

Although this is not a very long program, it makes three separate "trips" through the data: one to read the variables into a SAS data set, a second to compute the total income, and yet a third to create a set of recode variables. Simply by doing a little advance planning, you can greatly improve your SAS energy efficiency by performing all of these tasks in a single trip through your data:

```
data One;
/* Read data from external flat file */
infile Rawdata;
input id $char8.  (income1-income8) (9.);
/* Compute total income */
TotIncome = sum(of income1-income8);
/* Compute income recodes */
/* Recode variable calculations go here */
run;
```

AVOID UNNECESSARY SORTS

PROC SORT deserves special mention here. Certainly there are times when you need to sort your data, and when you do, PROC SORT is the way to go. Sorting does require another trip through your entire data set, however, and sorting large files can be extremely resource-intensive. Before you unnecessarily consume a large chunk of computer center resources, consider whether you need to sort your data at all.

For instance, if you need to get summary statistics for subgroups of a data set, you may be tempted to first sort the data set and then use PROC MEANS or PROC SUMMARY with a BY statement to do the grouping. But why make that extra trip, when you could use a CLASS statement instead and eliminate the need to sort your data?

The NOTSORTED option of the BY statement can be used to avoid unnecessary sorting when you have data that are *grouped* but not *sorted*. Perhaps you have a data set where records are grouped by state, but the states themselves are in order by region rather than alphabetically. If you need to use a BY statement to produce a grouped report, or to check on FIRST. and LAST. variables in a DATA step, you don't need to run PROC SORT on your data: use BY ... NOTSORTED instead.

```
proc print  data=CountyLevelData;
by state NOTSORTED;
var CountyName CountySeat CountyPopulation;
run;
```

Bear in mind that NOTSORTED is not a valid option in all situations–it won't work with PROC SORT, or with the MERGE or UPDATE statements.

You may also have a situation where you are creating a SAS data set from a non-SAS input file that you already know is sorted in the desired order. In such cases, there is no need to use PROC SORT. Instead, you can use the SORTEDBY data set option to tell SAS the order that your records are in. Using this option not only helps SAS to optimize certain operations, it also provides valuable internal documentation for permanent data sets.

```
data MySasData (SORTEDBY=ID);
infile rawdata;  /* We know this is already sorted by ID */
input id $char6. ... ;
run;
```

If you have a permanent data set where you frequently need to analyze just a single subgroup (perhaps records for a single state within the country), consider creating an index for the data set (see Raithel 2006), which will greatly improve speed and reduce resource consumption when you need to access a small subset of your records. Another advantage of indexing over sorting is that you can create any number of indexes for a single data set. For instance, you could efficiently analyze data from a single state in one run and analyze data for a single type of product in another run without having to restructure your data in between.

DON'T READ THE SAME SOURCE DATA AGAIN, AND AGAIN, AND AGAIN

This issue most often crops up when we have to extract and process a number of subsets from a large data set. We read the data set for the first time, extract the first subset, and then analyze the first subset. We read the data set for the second time, extract the second subset, and then analyze the second subset. Our SAS program continues on with extracting and analyzing additional subsets, each time reading the entire data set from the first record to the last record as it seeks to qualify records for inclusion in the subset. Though each of the subsets differs in content, they are each a result of reading the entire data set. So, extracting three subsets requires three passes through the data.

Here is an example of this type of approach:

```
data Africa;
set sashelp.shoes;
       where region eq "Africa";
run;

/*  other SAS code to specifically process Africa data set */

data Canada;
set sashelp.shoes;
       where region eq "Canada";
run;

/*  other SAS code to specifically process Canada data set */

data Pacific;
set sashelp.shoes;
       where region eq "Pacific";
run;

/*  other SAS code to specifically process Pacific data set */
```

In the example, the SHOES SAS data set is read in its entirety three times in order to extract three separate subsets and create three separate SAS data sets.

This is highly wasteful and inefficient when the source SAS data set is not indexed by the variable (or variables) in your WHERE statement, or when you are reading a flat file. It is inefficient because you are reading observations or records that you do not want again and again. Each pass of the entire SAS data set or flat file generates input/output operations (I/O's) which are the slowest event in the life of a SAS program. The more I/O's you generate, the longer your program runs; the fewer I/O's you generate, the quicker your program runs. If your source data set is stored on a network drive, you are creating unnecessary network traffic by constantly rereading the same data.

A more resource-friendly approach is to do the following:

```
data Africa Canada Pacific;
set sashelp.shoes;
       if region eq "Africa" then output Africa;
              else if region eq "Canada" then output Canada;
                     else if region eq "Pacific" then output Pacific;
run;

/*  other SAS code to specifically process Africa data set */

/*  other SAS code to specifically process Canada data set */

/*  other SAS code to specifically process Pacific data set */
```

In this example, the SHOES SAS data set is read only once, and three SAS data sets are created from that single pass. Afterward, the relevant portions of the program process each of the subsets.

This technique—a single pass of the source data set to extract all subsets—is especially important for reading flat files. Since flat files are not optimized for SAS processing, SAS has to do more work when reading them (such as converting numeric fields into its internal floating-point representation) than it does when reading in SAS data sets. So, you should read a flat file with a single pass to cut down on this additional overhead. Here is an example:

```
data Africa Canada Pacific;
infile shoes "C:\products\shoes.dat";

input @1  region     $25.
      @26 product    $14.
      @40 subsidiary $12.;

       if region eq "Africa" then output Africa;
              else if region eq "Canada" then output Canada;
                     else if region eq "Pacific" then output Pacific;
run;

/*  other SAS code to specifically process Africa data set */

/*  other SAS code to specifically process Canada data set */

/*  other SAS code to specifically process Pacific data set */
```

In this example, the SHOES.DAT flat file is read only once and three subset data sets are created for further analysis. This is more efficient than reading SHOES.DAT three times to create the Africa, Canada, and Pacific SAS data sets. This method gives you a single trip to the "data well" and ensures that you are not a SAS Energy Hog.

YOU MIGHT BE A SAS ENERGY HOG IF ... YOU MAKE A BIG DEAL OUT OF ADDING A FEW OBSERVATIONS

SAS Energy Hogs frequently find it frustrating to add a few observations to a large SAS data set, or to modify a few observations in a large SAS data set. They often use a DATA step to concatenate smaller SAS data sets to the large SAS data set. This results in elongated wait times, additional processor time, and additional I/O's across the network. All of that happens because SAS allocates a new copy of the large data set, reads the entire large data set, writes the large data set to the new copy, reads the smaller data set, writes each of its records to the end of the new copy of the large data set, deletes the original large SAS data set and renames the new large SAS data set to the name of the old large SAS data set.

Here is is an inefficient example of concatenating a small SAS data set with a large one:

```
data big_dataset;
set big_dataset
    small_dataset;
run;
```

A more efficient way to do this is to use the APPEND Procedure. The APPEND Procedure simply appends one SAS data set to another one, using one as the *base*—the data set being appended (concatenated) to. The observations in the base SAS data set are not read. Instead, SAS reads the observations from the second SAS data set and writes them to the end of the base SAS data set. Here is an example using PROC APPEND instead of the program above:

```
proc append base=big_dataset data=small_dataset;
run;
```

There are a number of obvious—and not so obvious—caveats with appending two SAS data sets in this manner. Most fundamentally, PROC APPEND will not change the *structure* of the base data set; so there are issues when one data set does not have a variable found in the other, or variables have different lengths in the two data sets. Before using this technique, check the SAS Online Documentation for PROC APPEND.

Adding, modifying, or deleting observations from a large SAS data set can be done efficiently using the MODIFY statement. The MODIFY statement allows you to modify a large SAS data set *in place*, without creating a new copy. A common way to use the MODIFY statement is to have a *transaction* SAS data set that contains a “key” variable that can be used to uniquely identify observations in the large *master* SAS data set. The *transaction* SAS data set contains observations that are to be added to the *master* SAS data set, or the key values of observations that are to be deleted from the *master* SAS data set, or observations with key values of observations in the *master* SAS data set and new values for some of the variables in those observations. Using the MODIFY statement, SAS reads the transaction SAS data set sequentially and matches observations in the master SAS data set via the key variables specified on the BY statement. Then, the action that you specify—add, replace, or remove—is done. Here is an example:

```
data indexlib.bigfile;
modify  indexlib.bigfile indexlib.tranfile;
by  sosecnum;

      if _iorc_ = 0 then do;      /* A match was found - update master */
            actual  = newactual;
            replace;
      end;
      else do;                      /* No match was found, add trans obs */
            actual  = newactual;
            output;
            _error_ = 0;
      end;
run;
```

In the example, *master* SAS data set **indexlib.bigfile** is updated by observations from *transaction* SAS data set **indexlib.tranfile** using the variable **sosecnum** as the key variable to find matches. Both data sets must be sorted by **sosecnum**. SAS sets the automatic variable _iorc_ to zero when a match is found. So, when _iorc_ equals zero, we update the value of **actual** in the *master* SAS data set to the value of **newactual** found in the transaction SAS data set and save the observation to the master SAS data set. When there is not a match, we create a new observation and write it to the master SAS data set. It is also necessary to change the _error_ automatic variable to zero to avoid warnings in the SAS log.

This next example uses a SAS index built from **sosecnum** to directly read observations from the master SAS data set. Note that there is no BY statement, so neither data set has to be sorted.

```
data indexlib.bigfile;
set  indexlib.tranfile;
modify  indexlib.bigfile key=sosecnum;

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

run;
```

As in the previous example, when there is a match on **sosecnum**, the value of actual 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 **sosecnum** was successful or not. This function should always be used to determine the success of modifying a SAS data set via an index.

The MODIFY statement is a powerful and complex tool for updating SAS data sets in place. Both Raithel (2006) and the SAS Online Documentation contain more detailed information about the MODIFY statement.

YOU MIGHT BE A SAS ENERGY HOG IF ... YOU DON'T USE THE MOST EFFICIENT TOOLS

You probably have a number of tools in your kitchen for cooking food, such as a regular oven, microwave, and perhaps a toaster oven. Since all of these devices use different amounts of energy, you can improve your energy efficiency by choosing the most efficient device that can properly cook a particular type of food.

USE PROC DATASETS TO CHANGE THE ATTRIBUTES OF VARIABLES

The same principle holds true in SAS. Have you ever created a SAS data set and later needed to change the attributes of some of its variables? You might have written code something like this:

```
data mylib.DemographicData;
set  mylib.DemographicData;
rename  sex = gender;
label   educ = '12-category education status recode';
format  zipcode z5.;
run;
```

At first glance, you might think this code is reasonably efficient–after all, you have managed to rename one variable, label a second variable, and apply a format to yet another variable, all in one SAS step. Before patting yourself on the back, however, consider that this program reads in and writes out every single record in your data set, which might not be very fast if you have millions of records and hundreds of variables. It certainly works, but can you save energy by using a better tool?

SAS data sets store "header" information, such as variable attributes, at the beginning of the physical file, followed by all of the actual data. Assuming that your data set is stored on a hard disk or some other direct-access device, it seems like it should be possible to change variable names, labels, and formats without having to reprocess the data at all. Fortunately, the MODIFY statement in PROC DATASETS allows us to do just that:

```
proc datasets  library=mylib;
modify DemographicData;
rename  sex = gender;
```

```
   label    educ = '12-category education status recode';
   format   zipcode z5.;
   quit;
```

This approach simply updates the attributes of the specified variables in the data set header without reading the data records, and therefore typically runs in a second or two–certainly a much more energy-efficient approach than reading and writing thousands or millions of records to do the same thing!

There are a few limits to this approach: you can't change the type (numeric vs. character) or length of variables using PROC DATASETS. Since these changes affect the underlying storage of your data, they *do* require SAS to read and rewrite all of your records. You also can't use PROC DATASETS to change the logical order of the variables in a data set.

USE SAS DATA SETS TO STORE YOUR DATA

The information that you need for your SAS programs may be stored in relational databases, flat files, Microsoft Excel spreadsheets, or in some other form. One of the most powerful features of SAS is its ability to read in (and write out) data from a wide variety of storage formats. In many cases, especially when SAS/ACCESS® engines are used, you (as the programmer) don't even need to worry very much about the underlying data storage mechanism. This transparency can be a big help when developing and maintaining your programs, especially when there is a change in how the data that you need are stored.

Despite the power of these interfacing capabilities, don't overuse them. In terms of efficiency, SAS is at its best when it's working with its own SAS data sets. (For some types of work, special SAS storage platforms such as the Scalable Performance Data Server may exceed the performance of traditional SAS data sets.) To avoid being a SAS Energy Hog, just keep two simple principles in mind.

First, don't pull non-SAS data into SAS more often than necessary. If you have some data in some non-SAS format (such as a flat file or an Excel spreadsheet), and you need to refer to it several times in your SAS program, read the data in once, save it in a temporary SAS data set, and then reference the SAS data set wherever you need to access the data.

Second, use SAS data sets for permanent data storage whenever possible. If you need to store files that will only be accessed from within SAS, SAS data sets are definitely the way to go in terms of efficiency, not to mention ease of program development and maintenance. Obviously there will be times when you will need to get data out of SAS for use in other software, but try to keep your data in SAS format as long as possible to maximize your SAS efficiency.

YOU MIGHT BE A SAS ENERGY HOG IF ... YOU DON'T HAVE YOUR OWN POINT OF VIEW

SAS Energy Hogs think that they have to create permanent SAS data sets for every occasion. They litter the disk space on their workstations and the disk drives on shared servers with SAS data sets, many of which contain initial or intermediate data and are not really needed in the long run. This practice wastes disk space and can lead to unnecessary disk space shortages, especially on shared servers.

Energy-conscious SAS programmers know that often a SAS Data View can be used instead of a permanent SAS data set. A SAS Data View is a type of SAS file that simply stores instructions for how to retrieve data from other files. Those files could be flat files, SAS data sets, DBMS tables, or other types of files. Along with instructions for obtaining data, views store descriptor information such as variable names, lengths, and formats.

A Data View has no actual SAS data (observations) stored in it, so it takes up virtually no space. When a SAS Data View is executed in a program, the instructions in the view access the source data sets, process and format the source data as per the information stored in the view, and feed the data as input to a SAS DATA step or a SAS Procedure. At no time does the view actually house observations.

SAS Data Views can be created with a SAS DATA step or with PROC SQL. This example illustrates using the DATA step to create a view. A SAS Energy Hog might write the following program:

```
        data premlib.extract01;
        merge input.bigfile  input.otherbigfile;

              by salesterr;
```

```
commission = .025 * sales;

<<other SAS code>>

run;

proc summary data=permlib.extract01;
       class orgnum;
       var sales commission;
output out=summ1 sum=;
run;
```

In the example, two very large SAS data sets are merged in a DATA step to create the **extract01** SAS data set. The **extract01** data set is even larger and thus takes up considerable space in the SAS data library. It only exists because the programmer needs merged data to feed into the Summary Procedure. Since there is no other need for this particular combination of data, this program could be recoded this way:

```
data permlib.extract01/view=permlib.extract01;
merge input.bigfile  input.otherbigfile;

       by salesterr;

commission = .025 * sales;

<<other SAS code>>

run;

proc summary data=permlib.extract01;
       class orgnum;
       var sales commission;
output out=summ1 sum=;
run;
```

The only difference between this example and the previous one is the VIEW= option on the DATA statement. Now, when the DATA step is executed, it does not process any observations. Instead, it accesses the descriptors of the **bigfile** and the **otherbigfile** SAS data sets and uses them, as well as the statements within the DATA step, to compile a data view named **extract01**. That view is stored in the **permlib** SAS data library for later use. When the Summary Procedure is executed, SAS executes the **extract01** data view. Doing so, SAS merges the **bigfile** and the **otherbigfile** SAS data sets and performs all of the SAS statements specified in the DATA step that was used to create the view. Data from the executing view is stored in memory and/or temporary tables and “streamed” into the Summary Procedure.

Since the **extract01** data view is stored in a permanent directory, it can be used by other SAS programs to send data from the merge of the **bigfile** and the **otherbigfile** SAS data sets to other DATA steps and procedures. And, it has a very small disk space footprint.

Saving disk space is not the only benefit of using a SAS data view. Other benefits include:

- SAS always uses a current version of the data sets input to the view, so data surfaced by the view is dynamic, whereas 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 from other users 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.

Energy-conscious SAS programmers know that there is always room for a view.

YOU MIGHT BE A SAS ENERGY HOG IF ... YOU DON'T KNOW THE MEANING OF THE WORD "ELSE"

Have you ever written SAS code that looks something like this?

```
if numvar = 0 then label = 'None';
if 1  <= numvar <= 10 then label = '1-10';
if 11 <= numvar <= 25 then label = '11-25';
if 26 <= numvar <= 100 then label = '26-100';
if numvar > 100 then label = 'Really big';
```

If so, you are definitely making SAS do much more work then necessary!

DON'T WASTE ENERGY WHEN MAKING COMPARISONS

Think about it for a minute. This code makes SAS test every single one of the conditional expressions. However, if numvar does in fact equal zero (the first condition), it can't possibly meet any of the other conditions. Therefore, you are definitely being a SAS Energy Hog by making it check all of the remaining conditions.

When only one of a set of conditions can be true, don't just put a series of IF statements in your code. Instead, you can use the IF ... ELSE IF ... ELSE IF ... ELSE structure to make your code more efficient. This coding technique stops SAS from unnecessarily testing subsequent conditions once one of them is satisfied:

```
if numvar = 0 then label = 'None';
else if 1  <= numvar <= 10 then label = '1-10';
else if 11 <= numvar <= 25 then label = '11-25';
else if 26 <= numvar <= 100 then label = '26-100';
else label = 'Really big';
```

(Note–we are assuming that numvar is always a nonnegative integer.)

That's the first step to improving our SAS energy efficiency in this type of situation. In this particular example, we can do even more. Notice how we are providing both a lower and an upper bound for numvar each time. However, given our knowledge that numvar is always a nonnegative integer, this is unnecessary–after the first comparison, we only need to check the upper limit:

```
if numvar = 0 then label = 'None';
else if ~~1  <=~~ numvar <= 10 then label = '1-10';
else if ~~11 <=~~ numvar <= 25 then label = '11-25';
else if ~~26 <=~~ numvar <= 100 then label = '26-100';
else label = 'Really big';
```

If you prefer, you can use a SELECT-block in these situations, rather than the IF ... ELSE IF ... approach:

```
select;
  when (numvar  = 0)   label = 'None';
  when (numvar <= 10)  label = '1-10';
  when (numvar <= 25)  label = '11-25';
  when (numvar <= 100) label = '26-100';
  otherwise            label = 'Really big';
end;
```

In this case, you should also consider whether this processing, which is essentially a table lookup, belongs inside your DATA step code at all. In most cases it's better to put this somewhere else, such as in a PROC FORMAT or in a lookup table whose values can be joined to the main data set.

YOU MIGHT BE A SAS ENERGY HOG IF ... YOU NEVER CLEAN UP AFTER YOURSELF

SAS Energy Hogs who run programs that create large intermediate SAS data sets in their WORK data library can find themselves running out of space on their C-drive, causing their programs to fail. Or, while running such SAS programs on shared servers (UNIX, Linux, Windows, etc.), they can end up soaking up all of the available WORK data library space, causing other SAS programmers' programs to fail. You can avoid these problems by using the good programming practice of cleaning up after yourself (programmatically speaking).

You can use the DATASETS procedure to delete intermediate SAS data sets once you no longer need them. Here is an example:

```
data file01;
set input.bigfile;

if salesterr = 27 and sales gt 1000000;

<<other SAS code>>

run;

proc summary data=file01;
       class orgnum;
       var sales;
output out=summ1 sum=;
run;

proc datasets library=WORK;
       delete file01;
run;

<<other SAS code>>
```

In the example, the **file01** SAS data is created in the WORK library and is a subset of the input.bigfile SAS data set. Once **file01** is summarized, it is not needed any more. Since it is a very large SAS data set, we have placed the DATASETS procedure into the program to delete **file01** from the WORK SAS data library, thereby freeing the disk space it occupies.

This is obviously a very simple example. You should use your best judgment to place PROC DATASETS at strategic points in your programs so that it judiciously removes either multiple obsolete data sets or large obsolete SAS data sets. Doing this will help you to make the best use of the WORK data library on your workstation, and to be a good neighbor to other SAS users when using a common WORK data library on your organization's servers.

YOU MIGHT BE A SAS ENERGY HOG IF ... YOU DON'T PLAY NICELY WITH OTHERS

Saving energy does not have to be a solitary activity. Car-pooling may be the best example of how we can work with others to save energy. Similarly, SAS does an excellent job of working with a wide variety of other software. Taking full advantage of these capabilities is an excellent way to improve the efficiency of your SAS applications.

MAKE YOUR RELATIONAL DATABASE WORK FOR YOU

The SAS system is particularly powerful in its capabilities for exchanging data with relational databases such as Microsoft SQL Server, Oracle, Sybase, and DB2. Over the years, the developers at SAS Institute have made this process much easier and more transparent than it used to be, so that in many cases you can write virtually your entire program without even thinking about whether your data are coming from SAS data sets or from a database. This is a major improvement in programmer productivity, since it lets you concentrate on the best way to solve your business problems rather than the nuts and bolts of data access. However, completely ignoring the fact that your data are coming from an RDBMS can produce programs that are not as efficient as they could be, particularly when your processing involves an extremely large number of records. While a complete discussion of efficiency when working with relational databases is beyond the scope of this paper (see the References below for additional resources), the tips below should help you be more efficient when accessing RDBMS data.

Naturally, your fundamental energy-saving techniques still apply when you are bringing data in from your relational database. Earlier in this paper, we discussed how carrying around extra variables and observations wastes resources. This is an even more important concept to remember when getting data from SQL Server or Oracle, since it is very likely that the RDBMS data will have to travel over your network to get from the database to the workstation or server where you are running SAS. Therefore, keeping unnecessary variables and observations wastes database and network resources, as well as those in your SAS session.

A frequent culprit in database processing is the use of SELECT * within PROC SQL. Putting this in your program is admittedly much more convenient than listing all of the variables you actually need, especially since the SELECT statement itself does not support SAS variable lists. An extremely useful workaround here is to utilize the DROP= or KEEP= data set options:

```
create table DesiredStudentVars as
select * from dbmslib.class (drop=age--weight);
```

As we previously discussed, remember that you maximize your program's efficiency by minimizing the number of times you bring non-SAS data into SAS. If you need the same set of data from your database several times within a single SAS program, bring it in once and save it to a temporary SAS data set (perhaps using a DATA step or PROC COPY), and then read the temporary SAS data set whenever you need the records. If you need the same data across multiple jobs, you can extend this strategy by extracting the data once into a permanent SAS data set, which can then be read by your other jobs.

Finally, you may be able to improve your efficiency by using your RDBMS to do data summarization tasks that you would normally do with SAS reporting procedures such as SUMMARY/MEANS, REPORT, TABULATE, or FREQ. By using a SELECT statement in PROC SQL, you have the database summarize the data and send *just* the results back to SAS, which will almost always be faster than forcing the database to send all the detail records over to SAS for it to summarize.

```
proc sql;
create table SummaryResults as
select sex, count(*) as _FREQ_
  from mydb.Mytable
  group by sex
  order by sex;
quit;
```

YOU MIGHT BE A SAS ENERGY HOG IF ... YOU DO ALL OF YOUR WORK DURING NORMAL BUSINESS HOURS

Even a SAS Energy Hog knows that the busiest time for the corporate network is during core business hours—usually 9:00am to 5:00pm. When running SAS jobs that move large amounts of data across a network—say from a network drive to your workstation—or running SAS jobs that process large amounts of data on a shared server, response time is slower because of the high volume of data traffic. Similarly, SAS programs that use large amounts of memory on shared servers compete for that memory with a lot more other tasks during the business day. As the number of concurrent SAS tasks—and other resource-intensive tasks—increases on the network or shared server, all users experience some degree of degraded response time, and your SAS programs take longer to run.

But, core business hours are "core business hours" for a reason. That is when most people are in the office working to process and analyze data, and to create deliverables for clients. Programs simply have to be run. Reports, graphs, and tables must be created, vetted and then delivered. What can an energy-conscious SAS programmer do?

Consider which parts of your SAS program you might be able to split out and run outside of your core hours. Typically, up-front, I/O intensive steps, such as extracting subsets from a DBMS or from large SAS data sets, lend themselves well for running in the evening, early morning, or over a weekend. Take a look at your SAS programs and determine whether or not the data extraction portion can be logically split out from the rest. If so, then:

- Create a separate SAS program that contains the data extraction.
- Create a batch file that will execute the data extraction SAS program.

- Schedule the batch file for off-hours.
    - The batch job will run and create permanent extract SAS data sets.
- Run the rest of your SAS program to process the extract SAS data sets during normal business hours.

Memory-intensive SAS programs, such as imputations and emulations are also good candidates for the off-hours. You can similarly create batch files that execute your program and then schedule them to run during the off-hours.

Most operating systems have facilities for creating batch files and scheduling them for execution:

- On Linux and Unix, you can create a batch script file and use the ***crontab*** to schedule its execution.
- On z/OS, you may use JCL to create a batch job and schedule it using your scheduling system (such as CA-7).
- On Windows, you can create a .bat file and use the Windows Scheduler to schedule it.

Splitting your SAS programs so that your most resource intensive tasks are done off-hours will help to lower your computer resource footprint during prime business hours. If enough of your coworkers follow suit, then there will be enough corporate computing power available during business hours to get the most urgent processing done.

CONCLUSION

Having read this paper, you should have a good idea as to whether or not *you* are a SAS Energy Hog. More than likely, you have already incorporated many—if not most—of the ideas presented in this paper into your SAS programming repertoire. Perhaps you found a few programming constructs or techniques that are new to you that you can use to improve the efficiency of your SAS programs. Or, maybe this paper got you to thinking about other ways of making your SAS programs more efficient in order to cut down on the computing resources that you consume. Whatever the case, you know that with a nominal amount of effort you can craft SAS programs that require the minimum amount of resources to get the maximum amount of work done. And, that will be good for you, your workmates, your organization, and—most importantly—your clients.

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

REFERENCES

Gorrell, Paul. 2007. “Numeric Length: Concepts and Consequences”. NorthEast SAS Users Group Inc. 20th Annual Conference Proceedings, Baltimore, MD.
Available: http://www.nesug.org/proceedings/nesug07/bb/bb05.pdf

Raithel, Michael A. 2006. *The Complete Guide to SAS Indexes*: Cary, NC: SAS Institute, Inc.
Available*:* http://www.sas.com/apps/pubscat/bookdetails.jsp?catid=1&pc=60409

Rhoads, Mike. 2008. “Avoiding Common Traps when Accessing RDBMS Data”. NorthEast SAS Users Group Inc. 21st Annual Conference Proceedings, Pittsburgh, PA.
Available*: http://www.nesug.org/Proceedings/nesug08/cc/cc01.pdf*

SAS Institute Inc. 2008. **SAS OnlineDoc® 9.1.3**. Cary, NC: SAS Institute Inc..
http://support.sas.com/onlinedoc/913/docMainpage.jsp

SAS Institute Inc. **SAS OnlineDoc® 9.2**
http://support.sas.com/cdlsearch?ct=80000

ACKNOWLEDGMENTS

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

Other brand and product names are trademarks of their respective companies.

The authors would like to thank Westat management for supporting their participation in SAS Global Forum 2009, as well as Duke Owen for his review and constructive suggestions. They would also like to thank each other for this rewarding collaboration.

CONTACT INFORMATION

The authors would love to hear your own SAS Energy Hog stories, whether they are a confession of wasteful past practices, ideas for additional SAS energy-saving techniques, or feedback on this paper. You can contact Mike or Michael by email:

Mike Rhoads – mikerhoads@westat.com

Michael Raithel – michaelraithel@westat.com