PhUSE 2010

Paper CC03

Working Efficiently with Large SAS® Datasets

Vishal Jain, Quanticate, Canterbury, United Kingdom

ABSTRACT

Many SAS users experience challenges when working with large SAS datasets having millions of rows, hundreds of columns and size close to a gigabyte or even more. Often it takes enormous time to process these datasets which can have an impact on delivery timelines. Also, storing such datasets may cause shortage of permanent storage space or can exhaust the available memory. In order to handle these constraints, one can think of making a large dataset smaller by reducing the number of observations and/or variables or by reducing the size of the variables without losing any of its information. This paper will focus on techniques supported by test results to reduce the size of huge datasets and work with them efficiently by striking a balance between the processing time and resource limitations. Some of the efficiency techniques described in this paper can also be applied while working with smaller datasets.

INTRODUCTION

A typical SAS dataset is made up of observations and variables. If we prefix “LARGE” before a SAS dataset, it implies that a SAS dataset may consist of numerous observations and variables thus resulting in increase in its overall size. A “LARGE” SAS dataset can be a relative or a subjective term as it primarily depends how a user perceives it and also on the available resources and storage space. I personally experienced working with large datasets while integrating or pooling the data from various studies. I often faced problems while storing and processing such datasets. It could take hours to read/retrieve data and use subsetting conditions to generate a report successfully. The whole idea behind this paper is mainly based on the challenges and constraints that were encountered while working on large datasets of around 1 GB size and the techniques that were used to efficiently manage the job within the available resources.

In the first section of this paper, I will endeavour to point out some advantages and drawbacks of the various dataset compression methods by demonstrating and comparing the results when tested on a large dataset. Depending on your project requirements and resource limitations you may be in a position to decide the appropriate method suitable for your purpose.

The second section of the paper will mostly talk about few other programming tips and techniques to make working with large datasets more efficient.

The SAS example codes mentioned in this paper have been tested on SAS v9.2 in UNIX environment. The results expressed may differ in other settings or operating environments.

SECTION I - REDUCING/COMPRESSING THE SIZE OF A DATASET

One can think of making a large SAS dataset smaller by using SAS statements like **LENGTH** or dataset compression tools like **COMPRESS** option to save the storage space. Reducing the size of the dataset can also help in reducing the time SAS takes to read or access data from a dataset.

A. LENGTH STATEMENT

A SAS variable can be either numeric or character. The number of bytes required to store variables in a SAS dataset can be set or controlled by LENGTH Statement.

Numeric variables in SAS are stored in default length of 8 bytes whereas for Character variables one byte corresponds to one character. It is not necessary that the complete length of the variable is utilized by its values. This is commonly observed during the creation of numeric dummy or flag variables whose only purpose is to hold either 0 or 1. Also, answers to questionnaires, categorical numeric variables and few character variables holding comments/long values may not accommodate whole of their allocated length. It becomes quite important to understand the way SAS stores values in its variables and identifying which variables can store the same information in reduced or compressed length. Reducing the variable length can help in reducing both the amount of storage space and the number of I/O operations required to read and write a dataset.

1. Numeric Variables

SAS stores numeric data using floating point representation. The following table specifies the largest integer that can be stored in SAS numeric variables in a specified length:

Length (Bytes)	Largest Integer represented on UNIX
3	8,192
4	2,097,152
5	536,870,912
6	137,438,953,472
7	35,184,372,088,832
8	9,007,199,254,740,990

As we can see from the above table, 8,192 is the largest integer that can fit into a length of 3 bytes on a UNIX system. Also, an 8 byte numeric variable can hold a very large integer value. Even though, one has a clear and detailed understanding of the study data, you cannot blindly specify a smaller byte length for the values of the numeric variables. Below are some of the points to remember before applying LENGTH compression on numeric variables:

i. Fractional numbers or numbers with decimals should be left with the default length of 8 as specifying a length less than the minimum required might result in a loss of accuracy due to truncation.
ii. Not more than 4 bytes of storage is required to store a reasonable SAS date value.
iii. No warnings or error messages are issued in the SAS LOG when the specified length in the LENGTH statement results in the truncation of data.

2. Character variables

For character variables, it becomes essential to know its longest value or the longest potential value in a particular variable. The length of each character variable can be set to the number of characters in the longest expected value of the variable by using LENGTH Statement.

Although it is possible to reduce the storage length of a variable using LENGTH statement, it requires additional programming time to perform length reduction and can potentially lead to incorrect results if inappropriately applied.

While applying length reduction approach on my SAS datasets, I came across an interesting macro **%SQUEEZE**, originally developed by Ross Bettinger (refer http://support.sas.com/kb/24/804.html) which finds the smallest length required by the numeric and character variables in a SAS data set and assigns the minimum lengths to these variables. This macro also offers some sort of flexibility by providing a macro parameter which can help in selecting the variables that you do not wish to compress.

I have tested this macro on few SAS datasets of various sizes and with different combinations of a number of numeric and character variables. I will show the test results later in this paper while discussing the SAS dataset compression tools. I will try to demonstrate the benefits and trade-offs of the COMPRESS=BINARY|CHAR option in SAS by using it on SAS datasets of different sizes and comparing its results with the compression results obtained from the LENGTH statement.

B. DATASET COMPRESSION USING COMPRESS OPTION

The length of an observation in a SAS dataset is the total length of the individual variables. Since a variable has the same length in each observation, all records in a SAS dataset occupy the same amount of storage space irrespective of whether there is any data in the record. Compression is a process that reduces the number of bytes required to represent each observation. Dataset options COMPRESS= BINARY|CHAR can be used to carry out the compression of observations in output SAS datasets. COMPRESS= NO disables compression. COMPRESS= CHAR can be used as an alternative to COMPRESS= YES in SAS Version 7 and later versions.

In theory, COMPRESS= BINARY option is highly effective for compressing medium to large (several hundred bytes or larger) blocks of binary data (numeric variables) whereas COMPRESS = YES|CHAR option would result in SAS attempting compression on repeating values or consecutive blanks in character variables. SAS LOG gives information of the percentage compression that occurred and also displays the number of pages of dataset that were saved by doing so. Fewer pages per dataset, the fewer I/O operations required.

A compressed dataset might require less storage space and fewer I/O operations to read from or write to the data during processing, however additional CPU resources are needed to access a compressed file and also there are few cases where compressing a SAS data set will result in a larger file being created, especially when the file has extremely short length records or there are no character variables with blank spaces to be compressed.

I have tested both COMPRESS= CHAR and COMPRESS= BINARY options on real life clinical trials datasets of various sizes ranging from 100 MB to 5 GB and observed that COMPRESS= BINARY proved to be more effective than the COMPRESS= CHAR in terms of reducing the storage space. Below are the results of the COMPRESS= option and LENGTH reduction using %SQUEEZE.

Original Data = 114.3 MB			
No. of Char Variables = 38		No. of Num variables = 7	
Length Reduction (Using %SQUEEZE)	**Resultant Size expressed as % of Original data size with/without COMPRESS Option**		
	No	COMPRESS= BINARY	COMPRESS= CHAR
No	0	21%	23%
Yes	40%	**19%**	21%

Original Data = 928.8 MB			
No. of Char Variables = 36		No. of Num variables = 19	
Length Reduction (Using %SQUEEZE)	**Resultant Size expressed as % of Original data size with/without COMPRESS Option**		
	No	COMPRESS= BINARY	COMPRESS= CHAR
No	0	13%	15%
Yes	24%	**11%**	13%

Original Data = 131.5 MB			
No. of Char Variables = 23		No. of Num variables = 42	
Length Reduction (Using %SQUEEZE)	**Resultant Size expressed as % of Original data size with/without COMPRESS Option**		
	No	COMPRESS= BINARY	COMPRESS= CHAR
No	0	16%	18%
Yes	25%	**14%**	17%

Original Data = 5.1 GB			
No. of Char Variables = 49		No. of Num variables = 90	
Length Reduction (Using %SQUEEZE)	**Resultant Size expressed as % of Original data size with/without COMPRESS Option**		
	No	COMPRESS= BINARY	COMPRESS= CHAR
No	0	16%	22%
Yes	20%	**14%**	17%

The above tables show that LENGTH Statement when used in combination with COMPRESS= BINARY option yields the best results with respect to reduction of data size and its storage. This approach helped me to achieve an average compression of around **85 %** in my SAS datasets. The size of the resultant data was around **15 %** of the original data size.
Few SAS programs were run on these reduced datasets and it was observed that the performance was faster than running them through corresponding original datasets. This observation may not be true with all SAS datasets as it largely depends on the size and structure of the dataset, number and lengths of the variables, SAS job to be performed and the operating environment. I would suggest the reader to evaluate each of the compression techniques on their data before application.

C. SIZE REDUCTION BY SELECTION OF VARIABLES

A SAS dataset may contain many variables that are either completely blank or not required for report generation. Dropping such variables does not make a huge difference when the size of the dataset is small. But, for large datasets, eliminating such missing variables will definitely improve the efficiency of the program. Also, it is not always required to have all variables to be present in the input dataset of a SAS Procedure. The most common case is while processing a SAS dataset with a procedure to calculate frequency counts or basic summary statistics. It is recommended to exclude any such variables that are not needed.

The **DROP=** and **KEEP=** option or DATA step statements **DROP and KEEP** can be used to select variables from a SAS dataset. Variables with all missing values can be determined during DATA Step processing. Also, there are several programs

available online which can identify and remove any character or numeric variables that have only missing values. One can refer to the following links for more details:
http://support.sas.com/kb/24/612.html or http://support.sas.com/resources/papers/proceedings10/048-2010.pdf

The table below will show us how dataset options (DROP= and KEEP=) are flexible, advantageous and efficient to use over DROP and KEEP DATA step statements:

KEEP= and DROP= dataset Options	**KEEP and DROP DATA step Statements**
Can be exclusively used to affect either input and/or output data wherever these options are mentioned	Always applied to the data being output
More efficient as they can be applied to a variable before being read from source data.	Less efficient as they apply only to the output dataset being written.
SAS may issue an ERROR if a variable specified using these options does not exist (assuming the default value of System Options DKRICOND = or DKROCOND = is not altered)	SAS will issue a WARNING instead of an ERROR message in the log if the variable does not exist. This can potentially lead to incorrect results if the same variable is referenced further in the program.
Can be used in Procedures	Cannot be used in Procedures

Until now, we have discussed how LARGE datasets can be made smaller to help us improve the program efficiency by saving processing time and storage space. In the next section of the paper, I will focus on how we can add further programming efficiency through subsetting, sorting and reducing disk space while working with these datasets.

SECTION II - PROGRAMMING EFFICIENCY

A. SAVING STORAGE SPACE OR MEMORY

1. While working on analysis datasets, one usually tends to create flag variables based on certain criteria or group observations with similar characteristics. Such variables may not be used to perform any arithmetic calculations. Since, the default length of numeric variable is 8 bytes and its minimum length cannot be less than 3 bytes, instead one can think of creating a character variable to store such flags. For instance, if a flag contains either 0 or 1, then it can be stored as a character variable having a length of 1 byte.

2. One of the most important tips that can help us to efficiently utilize the available storage space is to delete the SAS datasets in the WORK library, other temporary space, or permanent space when we no longer need them. This can be done using the **DATASETS** procedure as shown in the example below:

   /* To clear all work datasets */

   ```
   PROC DATASETS LIBRARY=WORK KILL NOLIST;
   QUIT;
   ```

3. One can also use **_NULL_** as a dataset name in the DATA statement when we need to execute a DATA step but do not want to create a SAS dataset. This is particularly useful while creating macro variables based on values obtained from an input SAS data set. For example:

   /* To count number of observations in dataset */;

   ```
   DATA _NULL_;
    SET raw.testdrug END =eof;
     IF eof THEN CALL SYMPUT('nobs', left(put(_N_)));
   RUN;
   ```

4. **SAS Views** are virtual SAS datasets that can be used as an alternative to SAS datasets. SAS Views contain only the descriptor information about the data and instructions for accessing or retrieving the data values. Views can be created in SAS with either PROC SQL or the DATA Step. For example,

   ```
   DATA demog / VIEW = demog;
    SET raw.demog;
   RUN;
   ```

View-name should match the dataset name specified in the DATA statement.

SAS Views can be used to avoid unnecessary reading or writing of temporary datasets and occupy very small amount of storage space as compared to the space required by a SAS data set. The example below shows subsetting results with and without using SAS views:

/* Subsetting a dataset without using SAS View */

```
DATA ecg;
  SET raw.ecg;
   WHERE visit ne 20;
RUN;

NOTE: DATA statement used (Total process
time):
      real time           1:29.92 seconds
      cpu time              32.20 seconds
```

- It occupies 595.3 MB WORK space.

/* Subsetting a dataset using SAS View */

```
DATA ecgview/ VIEW= ecgview;
  SET raw.ecg;
   WHERE visit ne 20;
RUN;

NOTE: DATA statement used (Total process
time):
      real time           0.28 seconds
      cpu time            0.07 seconds
```

- It occupies 24 KB WORK space.

Another example below shows how a large Lab Comments dataset is merged with treatment information (a small dataset having unique records per subject) using SAS Views. We can safely assume that both the datasets are already sorted by subjid (common variable).

/* Creating a SAS view for a large dataset */

```
DATA comments / VIEW = comments;
 SET raw.comments;
RUN;
```

/* Merging the SAS view with treatment information */

```
DATA combine;
   MERGE trtdrug(IN=a) comments(IN=b);
    BY subjid;
 RUN;
```

The SAS View “comments” occupies almost negligible work space as compared to its work dataset that could have been created without using VIEW = option. This VIEW is further merged with the treatment data to create a Combine dataset with all the required information.

Although, SAS VIEWS can reduce a large amount of disk space, it takes additional time to process the data defined by a SAS VIEW as compared to processing a regular SAS dataset.

5. In order to join two or more datasets, either **SQL JOIN** or **DATA Step MERGE** can be used depending on your data situation. Since the data needs to be sorted first before using DATA Step MERGE, one tends to create temporary datasets in WORK library. SQL can help eliminating such requirement and thus saving the storage space.

B. SUBSETTING OBSERVATIONS

It is recommended to perform data subsets early in the code. By subsetting or filtering any unwanted records from a dataset, we can reduce the size of the data thus saving storage space and improving efficiency of the program. For this purpose, either a **WHERE** statement or an **IF** statement can be used. Although both are useful for limiting the number of observations in a dataset, one needs to understand the differences between the two in terms of programming efficiency. Below table shows that depending on the underlying task and data situation, one can determine the appropriate utility of these statements:

IF Statement	WHERE Statement
Subsets after processing or merging the data	Subsets a dataset before processing or merging of data
Can be used only within a DATA step	Can be used in DATA step and/or Procedures

Apart from the dataset variables, it can also be used when specifying the automatic variables like _N_, FIRST.BY and LAST.BY or any new variables created within the DATA step	Can be used for specifying only the dataset variables
Can be used to conditionally execute the statement	Cannot be used for conditional execution
SAS automatically converts variable types with IF conditions (e.g. if a character value is specified instead of numeric in IF condition, no ERROR is generated by SAS)	It generates an ERROR as WHERE Clause operator requires compatible variables

If we only require to subset a large dataset based on a specific criteria or condition, then a WHERE statement in most of the cases proves to be efficient and faster in performance than an IF statement. In the example below, IF and WHERE statements were tested for performance on an ECG dataset of size around 715 MB with more than 446K observations.

/*Using IF Statement to keep only non-baseline records*/

```
DATA ecg;
  SET raw.ecg;
  IF visit ne 20;
RUN;

NOTE: DATA statement used (Total process
time):
      real time           29.30 seconds
      cpu time            21.65 seconds
```

/*Using WHERE Statement to keep only non-baseline records*/

```
DATA ecg;
  SET raw.ecg;
  WHERE visit ne 20;
RUN;

NOTE: DATA statement used (Total process
time):
      real time           14.93 seconds
      cpu time            14.30 seconds
```

Both the statements are doing the same task of subsetting the data when VISIT is not equal to 20. But we can easily conclude from the difference in the total process time that WHERE statement has a better performance than IF statement.

C. SORTING THE LARGE DATA SETS EFFICIENTLY

Sorting a large dataset with several key variables can take enormous time and it can cause insufficient disk space or memory error.

In order to subset a dataset with specific condition and then sort it using some key variables, many SAS programmers prefer the traditional way as mentioned below:

```
DATA vitals;
  SET vad.vitals(WHERE= (visit ne 'BASELINE'));
RUN;

PROC SORT data = vitals
          OUT  = novitbl;
  BY subjid vsdt;
RUN;
```

But the above approach seems to be less efficient when used on large datasets. An alternative to perform this task better can be:

```
PROC SORT data = vad.vitals(WHERE= (visit ne 'BASELINE'))
          OUT  = novitbl;
  BY subjid vsdt;
RUN;
```

When the above method was tested on a dataset of around 715 MB size, it was observed that the total process time (CPU & Real time) taken for subsetting and sorting in the same procedure is lesser than total time consumed to individually subset a data and then sort it in two separate steps. Also, only one dataset was created in the WORK library instead of two (as in traditional approach), thus saving the storage space.

Another method that can prove useful to overcome the barrier of insufficient disk space while sorting a large dataset is using PROC SORT with TAGSORT option as below:

```
PROC SORT data=vitals TAGSORT;
  BY subjid vsdt;
RUN;
```

When **TAGSORT** option is used, only sort keys (BY variables) and the observation number (identifier) for each observation are stored in the temporary files. The BY variables, together with the identifiers, are referred to as “tags”. At the completion of the sorting process, these tags are used to retrieve the records from the input data set in sorted order.
If your data is short and wide i.e. the total byte length of the BY variables is small as compared to the length of the record, then you may see an improvement in sorting speed by using TAGSORT and temporary disk usage might reduce considerably. For example, in order to sort a Treatment dataset of around 908 MB size, the following results were observed without or with using TAGSORT.

/* Sorting without TAGSORT option */

```
PROC SORT data = raw.testdrug
          OUT  = work.trt;
 BY study subjid;
run;

NOTE: PROCEDURE SORT used:
      real time    1:23:21 seconds
      cpu time     1:09:37 seconds
```

/* Sorting using TAGSORT option */

```
PROC SORT data = raw.tesdrug
          OUT  = work.trttag TAGSORT;
 BY study subjid;
run;

NOTE: PROCEDURE SORT used:
   real time    28.28 seconds
   cpu time     27.66 seconds
```

I have also observed that in some cases (for instance, when data is small) there is either increase or no reduction in the processing time. I recommend testing this option first on your data before using.

D. OTHER CONSIDERATIONS

1. If any two datasets contain the same variables and the variables possess same length and type, then use **PROC APPEND** for appending instead of the **SET** statement. The APPEND procedure concatenates much faster than the SET statement, particularly when the BASE= data set is large, because the APPEND procedure does not process the observations from the BASE= data set.

2. When only one condition can be true for a given observation, instead of using series of IF-THEN statements, one can write a block of **IF-THEN/ELSE** statements. The IF-THEN-ELSE statement conditionally executes a SAS statement depending on whether the given expression or condition is true or false. Using only IF-THEN statements causes SAS to evaluate all IF-THEN statements. Using IF-THEN statements with the ELSE statement causes SAS to execute IF-THEN statements until it encounters the first true statement. Subsequent IF-THEN statements are not evaluated.

3. It is suggested to test the SAS code first on smaller datasets by splitting the large SAS datasets into smaller manageable units. Logically breaking up the large datasets can assist in smoother and convenient data-handling. For example, a typical large laboratory dataset can be split into different categories such as Hematology, Chemistry, Urinalysis etc.

CONCLUSION

The main objective of this paper was to make the reader aware of the challenges and constraints that are encountered while working with “LARGE” SAS datasets. There are many methods available to overcome these challenges. But it is very important for us to be aware of the advantages and trade-offs of each of these techniques depending on our data situation, project requirements and available resources.

I would strongly recommend the reader to test and evaluate each of the methods discussed in this paper on their own study data before deciding the best approach. Ultimately, all the discussions boil down to EFFICIENCY. A piece of SAS code can only be termed “Efficient” if it can appropriately balance the code development time, I/O, processing time and resource and memory usage.

REFERENCES

SAS Support, http://support.sas.com/documentation/

Paul Gorrell, NESUG 2007, Numeric Length: Concepts and Consequences

Andrew H. Karp and David Shamlin, SUGI Paper 3-28, Indexing and Compressing SAS® Data Sets: How, Why and Why Not

Sunil Gupta, SAS Global Forum, Paper 213-2007, WHERE vs. IF Statements: Knowing the Difference in How and When to Apply

Selvaratnam Sridharma, NESUG 2006, How to Reduce the Disk Space Required by a SAS® Data Set

SAS Support, Sample *24804:* %SQUEEZE-ing Before Compressing Data, Redux

CONTACT INFORMATION

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

Author Name	: Vishal Jain
Company	: Quanticate International Ltd.
Address	: Floor 2, Exchange House Lakesview Business Park, Hersden – Canterbury, Kent, United Kingdom.
Postcode	: CT3 4NH
Work Phone	: +44(0) 1227 714 074
Fax	: +44(0) 1227 714 061
Email	: vishal.jain@quanticate.com
Web	: www.quanticate.com