SCSUG 2015 – Paper DM-06

Sorting a Bajillion Records: Conquering Scalability in a Big Data World

Troy Martin Hughes

ABSTRACT

"Big data" is often distinguished as encompassing high *volume*, *velocity*, or *variability* of data. Although big data can signal *big* business intelligence and *big* business value, it also can wreak havoc on systems and software ill-prepared for its size. *Scalability* describes the ability of a system or software to meet the needs of additional users or its ability to utilize additional processors or resources to fulfill those added requirements. Scalability also describes the adequate and efficient response of a system to increased data throughput. Because sorting data is one of the most common (as well as resource-intensive) operations in any programming language, inefficiencies or failures caused by big data are often first observed during sorting routines. Much SAS® literature has been dedicated to optimizing big data sorting for efficiency, including minimizing execution time and, to a lesser extent, minimizing resource usage (i.e., memory and storage consumption). Less attention has been paid, however, to implementing big data sorting that is reliable and robust even when confronted with resource limitations. To that end, this text introduces the SAFESORT macro that facilitates *a priori* exception handling routines (that detect environmental and data set attributes that could cause process failure) and *post hoc* exception handling routines (that detect actual failed sorting routines). If exception handling is triggered, SAFESORT automatically reroutes program flow from the default sorting routine to a less resource-intensive routine, thus sacrificing execution speed for reliability. However, because SAFESORT does not exhaust system resources like default SAS sorting routines, in some cases, it performs more than 200 times faster than default SAS sorting methods! Macro modularity moreover allows developers to select their favorite sorting routine and, for data-driven disciples, to build fuzzy logic routines that dynamically select a sort algorithm based on environmental and data set attributes.

INTRODUCTION

In the fire and rescue service, a mass casualty incident is often defined as any event that exceeds the current resources of the response. Thus, a multi-vehicle accident in one jurisdiction might thoroughly overwhelm resources while elsewhere this could be a common occurrence easily handled with sufficient apparatuses and rescue crews. Big data represent a similarly amorphous moving target, in that big data to one team or organization might not represent big data to another, because the volume, velocity, or variability are not overwhelming or even noteworthy. Notwithstanding, all systems have resource limitations that should be understood to build software that executes reliably and efficiently. To prepare for mass casualty incidents, fire departments have formal reciprocity agreements to ensure adequate resources can be summoned immediately when necessary. SAS software that is required to be reliable should also expect big data and ensure that processes will scale appropriately when confronted with increased data throughput.

Sorting data is a familiar objective in data analytics and a common prerequisite for many data analytic activities. The versatility of Base SAS provides several alternative methods and options for performing a data sort, including the SORT procedure, the ORDER BY statement in the SQL procedure, and the hash object. In some cases, the observations themselves do not need to be sorted but instead can be optimized and accessed through a file index. Each of these techniques has pros and cons that are described extensively in SAS literature, and which will vary the procedure execution time, memory consumption, and disk space usage. Thus, SAS practitioners should select the specific sort routine that best suits their environment, data set, and sorting objective rather than using a cookie-cutter approach to all sorting tasks.

Before comparing or selecting a sorting methodology, it's necessary to understand the ultimate objective and whether a sort is actually required. For example, must the observations be sorted, or will an index suffice that merely references the sorted order? In other cases, the sort can be combined with other processes in a single step. For example, a common objective is to individually sort two data sets, after which they are merged into a single data set, sometimes referenced in SAS literature as a SORT-SORT-MERGE. A multi-step SQL procedure, the FORMAT procedure, or a hash object can sometimes perform the SORT-SORT-MERGE with greater efficiency than individual SORT procedures

followed by the MERGE statement. Often only through trial and error can the most efficient solution be achieved to sort a specific data set in a given environment.

While efficiency is one of the most commonly sought requirements of software performance, it can be described and measured a number of ways. Software runtime is the most readily assessed efficiency metric, and code that executes twice as quickly while producing identical functionality (as other code) is sometimes said to be more efficient than its counterpart. The measure is intuitive, straightforward to collect, and can be assessed by developers and non-developers alike. Not surprisingly, much of SAS literature references *efficiency* in terms of expedited runtime and, for many purposes, this is both a valid and sufficient definition. But *efficiency* also must take into account the software's use of resources, including memory consumption and disk usage. Thus, if a process executes quickly but monopolizes system resources to the extent that other programs slow or fail to run, by some standards and depending on requirements of the software, it could be said to be *inefficient*.

One irony about software efficiency is that while execution time may be its most well-known and attributable metric, execution time is also the efficiency attribute least responsible for process failure. Yes, business value can be lost or eliminated if software executes too slowly, but this slowness typically does not cause sort routines to terminate in error. Memory faults and running out of disk storage, however, do cause runtime errors, yet in many SAS environments these attributes are neither monitored nor modified programmatically. Thus, where software requirements specify that sorting routines must be reliable and robust inasmuch as they should execute swiftly, resource constraints must be monitored and software must adapt when resource limitations cause process failure.

Some memory or disk storage errors can be predicted in sorting routines based on factors associated with higher resource usage. For example, increasing *ad infinitum* the number of observations, variables, sort variables, variable lengths, or file size will cause process failure in a sorting routine. The SAS infrastructure, operating system (OS), network, and specific sorting algorithm also influence the speed and success of sorting routines. Thus, a SAS sorting process optimized for a high-memory environment may function efficiently in that environment but fail when executed on other systems with lower memory levels. Within each specific SAS environment, developers should be able to estimate big data conditions that can portend process failure, such as the data set size and sort complexity. Thus, a priori exception handling routines can be used to detect these conditions and reroute program flow away from a sort that will likely fail. Post hoc exception handling routines, on the other hand, detect errors that occur during a sorting routine, and thus are able to validate sorting success or to initiate low-resource sorting when failure does occur.

The macro SAFESORT, included in Appendix A, provides an alternative, less resource-intensive sorting framework that can facilitate slower yet more reliable sorting when primary sorting routines fail or are predicted to fail due to big data or other resource limitations. SAFESORT utilizes a *divide-and-conquer* algorithm that iteratively sorts segments of the data set into chunks and ultimately joins these in a single, sorted output data set. Modular design facilitates swapping of sort routines, enabling developers or automated processes to assess data set attributes and environmental conditions to select the most optimized sort routines. For SAS practitioners who are experiencing inefficient and unreliable data sorting due to big data or resource limitations, SAFESORT offers a programmatic solution that dramatically extends native Base SAS sorting capabilities.

SAMPLE DATA SETS

To perform stress testing for sorting algorithms, tens of thousands of sort routines were iterated across thousands of data sets. To standardize this process, all data sets (unless otherwise specified) contained five character variables (each 20 characters in length and filled with 20 random upper case letters) and five numeric variables (each filled with numbers ranging from 0 to 9,999,999,999). Because no missing data were present, data set size was nearly perfectly correlated with number of observations, as depicted in Figure 1, which demonstrates the file size of data sets ranging from 100,000 to 4 million observations.

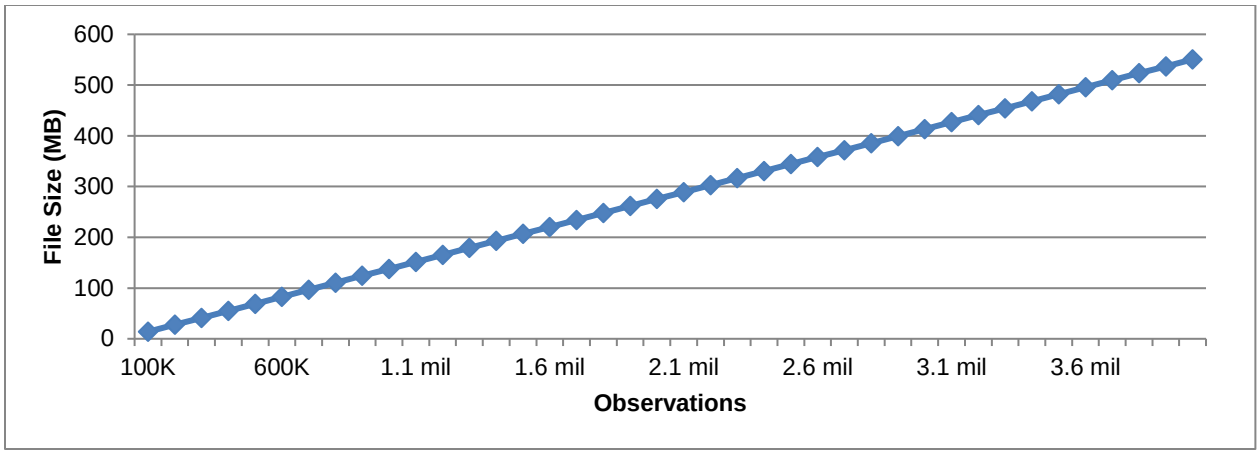

Figure 1. Correlation Between Number of Observations and File Size of Data Sets

The TOWEROFBABEL macro, included in Appendix B, produced all sample data sets and allows the user to vary the number of character variables, number of numeric variables, and number of observations to produce an endless array of systemically modified sample data sets. Results presented throughout this text are meant to illustrate programmatic factors that influence SAS sorting efficiency and failure, but should not be generalized to all data sets, situations, or SAS environments. For example, Figure 3 depicts the reduced efficiency of the hash object over time given a specific data set and memory reserve, but should not be misconstrued to represent performance of the hash object in all environments. Rather, the figure is meant to show that at some point on every system the hash object—like all sorting methods—will decrease in efficiency due to resource limitations and will eventually fail.

FACTORS AFFECTING SORTING EFFICIENCY

Sorting smaller, more manageable data sets may require no attention to efficiency or resource consumption because a process can complete in seconds regardless of how well or poorly it is coded. In fact, poor coding habits can flourish in environments in which data management and analysis are not restricted by system resources. As data sets increase in size or resources become limited, however, sorting scalability must be maintained to ensure that reliability and efficiency do not suffer. One author thoroughly discusses a number of best practices that elicit higher performance from the SORT procedure.[i] Some options such as deleting unnecessary variables or observations should also be considered and can demonstrate significant performance gains.[ii] The *SAS Scalability and Performance Community* website also describes methods and best practices that can deliver scalable solutions.[iii]

While sorting *reliability* rather than *efficiency* is the focus of the SAFESORT macro, the underlying factors that contribute to sorting performance and efficiency must be understood to facilitate the minimization of resource utilization. An efficient sorting algorithm can forestall the point at which the sort will fail due to resource limitations and thus the point at which a low-resource sort must be implemented. Several factors affect sorting speed and efficiency, some of which can be controlled programmatically and others of which are dependent on the SAS system, hardware, or network.

- **Available Memory** – Some sorting algorithms such as the hash object are more memory-intensive than others and thus can benefit more from increased memory. Decreased memory levels can cause sorting routines to perform slowly and, if memory levels fall too low, to fail outright. Figure 2 depicts the memory usage (obtained from the FULLSTIMER option Memory metric) of the hash object and SORT procedure when sorted by two numeric variables in the sample data set.

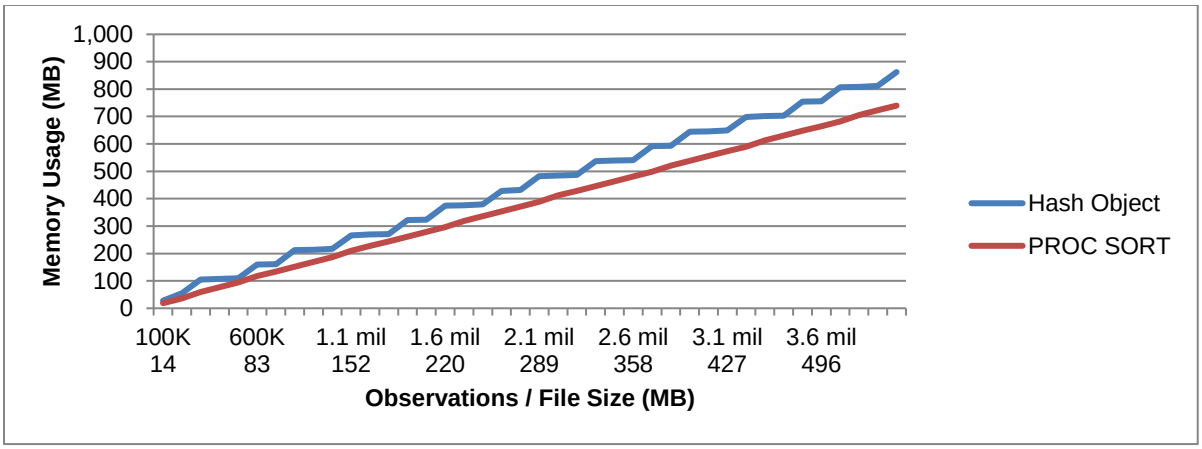

Figure 2. Memory Usage of SORT Procedure and Hash Object

Despite both sorting methods demonstrating memory consumption that is roughly linear to file size, as the hash object approaches the sorting memory threshold of 1 gigabyte (GB), runtime increases dramatically, as depicted in Figure 3. Examining FULLSTIMER metrics at 4 million observations for the hash object further reveals that the system memory peaked at 936 megabytes (MB), utilizing nearly all available resources and explaining the source of the tremendous inefficiency.

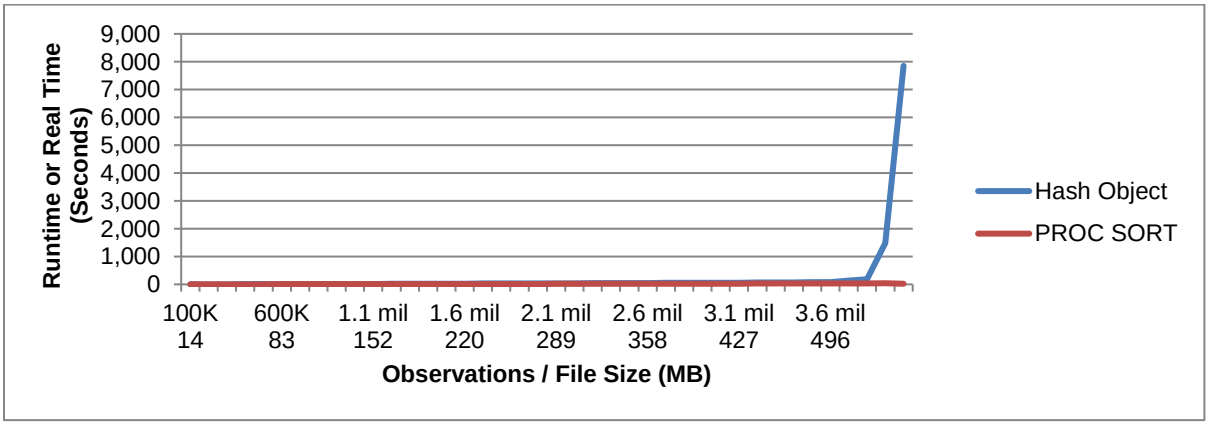

Figure 3. Runtime of SORT Procedure and Hash Object

The rapid acceleration of performance loss occurs at some point for all sorting methods as they begin to use all available resources. Figure 4 depicts the SORT procedure iterated from 100,000 to 5 million observations and sorted by one numeric variable in the sample data set. At 4.7 million observations, the sort completes in only 56 seconds, while at 4.8 and 4.9 million observations, it completes in eight minutes and 2.7 hours, respectively.

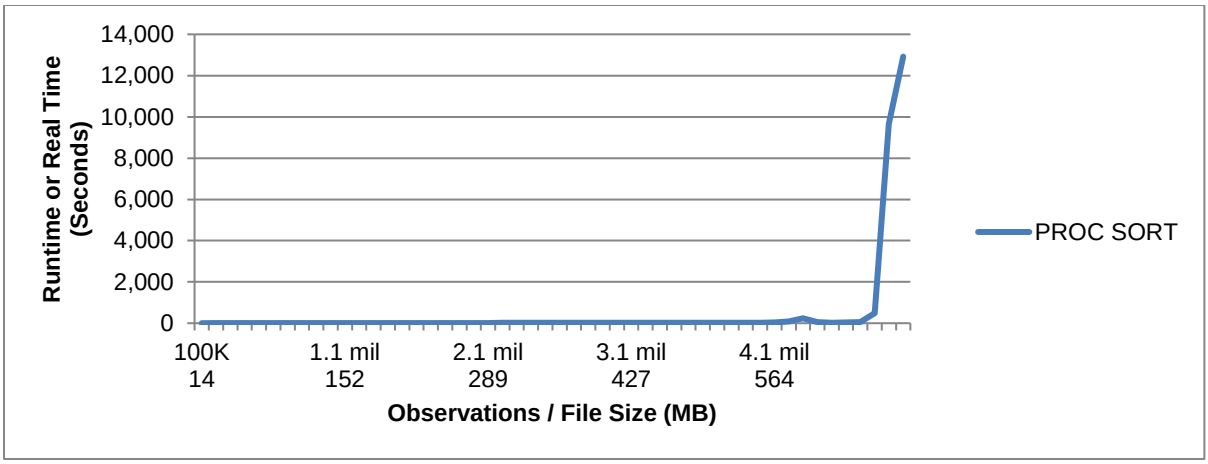

Figure 4. Runtime of SORT Procedure

Like the hash object, the SORT procedure ultimately and dramatically loses efficiency as memory limitations are approached. Figure 5 depicts memory usage of the previous SORT procedure (from the FULLSTIMER Memory metric) and demonstrates that as the SORT procedure slowed dramatically around 4.8 million observations, 848 MB of memory and 877 MB of system memory (not shown) were used.

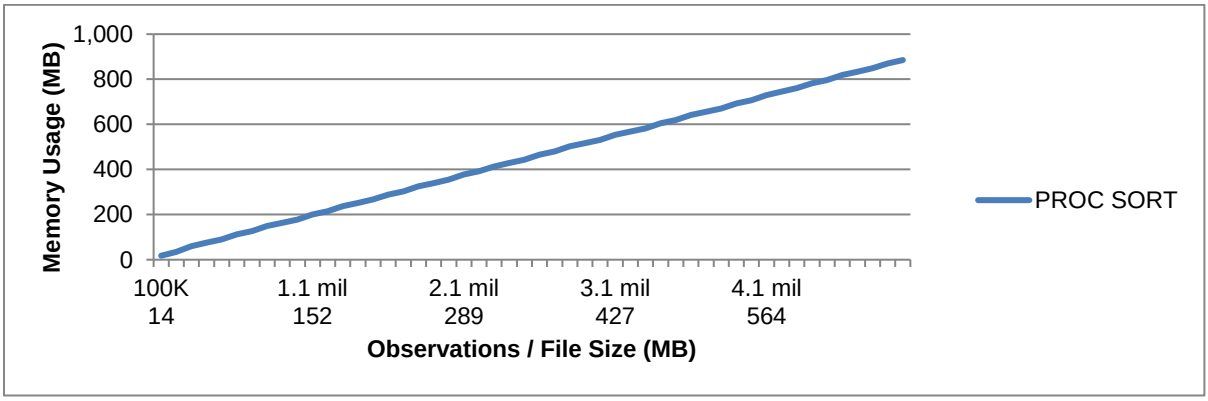

Figure 5. Memory Usage of SORT Procedure

- **Multithreaded Processing** – Beginning with Base SAS 9, the SORT procedure features default multithreading. [iv] Additional CPUs deliver increased processing power and quicker results.[v] The system option CPUCOUNT describes the number of CPUs that are in use and can be modified to the maximum value with the ACTUAL option, as demonstrated in the following code. While sorting with the CPUCOUNT=2 option will perform noticeably faster than CPUCOUNT=1, increasing the CPUCOUNT beyond two often unfortunately offers no further performance improvements.

```
options cpucount=actual;
%let cpucount=%sysevalf(%sysfunc(getoption(cpucount)));
%put &cpucount;
```

With multithreading disabled, the CPUCOUNT option has no effect, thus a CPUCOUNT of 1 is equivalent to executing the SORT procedure with the NOTHREADS option implemented. Figure 6 depicts the equivalent execution times demonstrated by CPUCOUNT=1 and NOTHREADS options, both of which eliminate multithreading. The PROC SORT function depicts the de facto sort method with multithreading enabled on two processors. The CPUCOUNT=1 and NOTHREADS options are not discussed further in this text due to their inherent reduced performance.

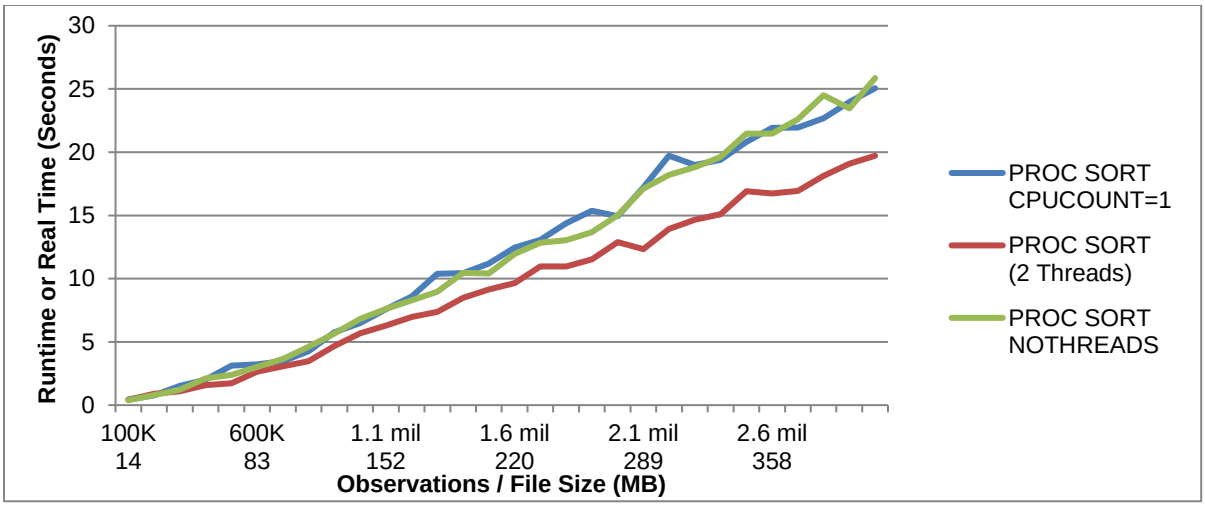

Figure 6. Runtime of SORT Procedure Showing Multithreading Outperforming Single-Threading

- **Sorting Algorithm** – The sorting method or algorithm, including use of options such as COMPRESS or TAGSORT, can have a profound and often unintended effect on efficiency. Some developers, teams, or organizations prefer one method over another, potentially excluding more efficient methods to their detriment. Figure 7 depicts execution time for sorting with the SORT procedure, SQL procedure, and hash object, iterated from 100,000 to 1 million observations in the sample data sets, and sorted by one character variable. This is not meant to demonstrate that one method will consistently outperform others, but rather that noticeable differences in performance can exist and should be investigated.

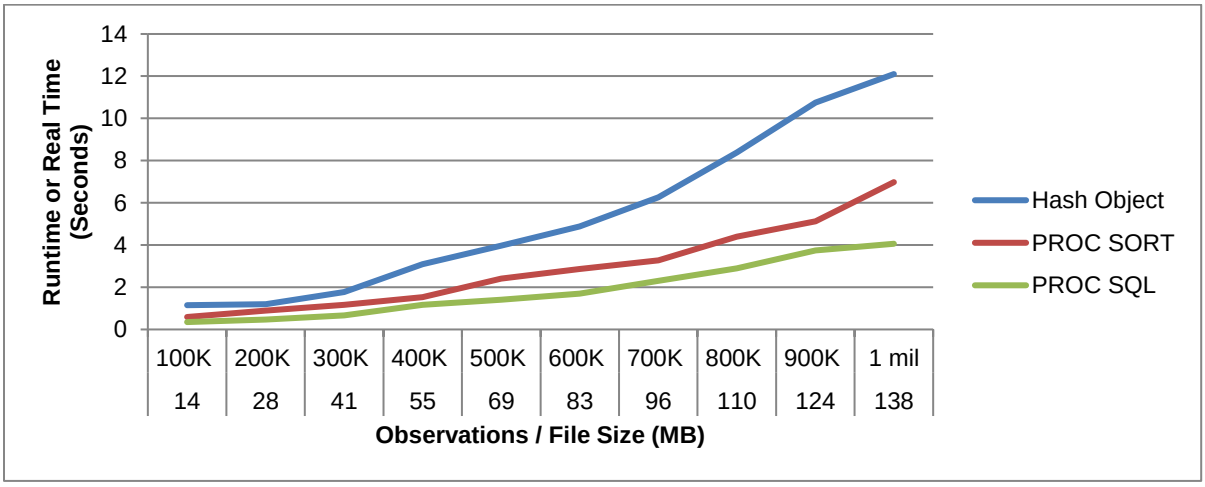

Figure 7. Runtime Comparison of Basic Sorting Methods

- **File Size** – Of all data set attributes, file size most accurately predicts sort execution time and use of resources, with larger data sets requiring increased execution time and resources. This relationship is demonstrated throughout this text and typically is linear but, as resources become depleted, the function can rapidly accelerate toward inefficiency, as depicted in Figures 3 and 4.

- **Number of Sort Variables** – Increasing the number of sort variables increases memory usage, as depicted in Figure 8, which iterates across data sets ranging from 100,000 to 2 million observations and sorts by between one and five character variables.

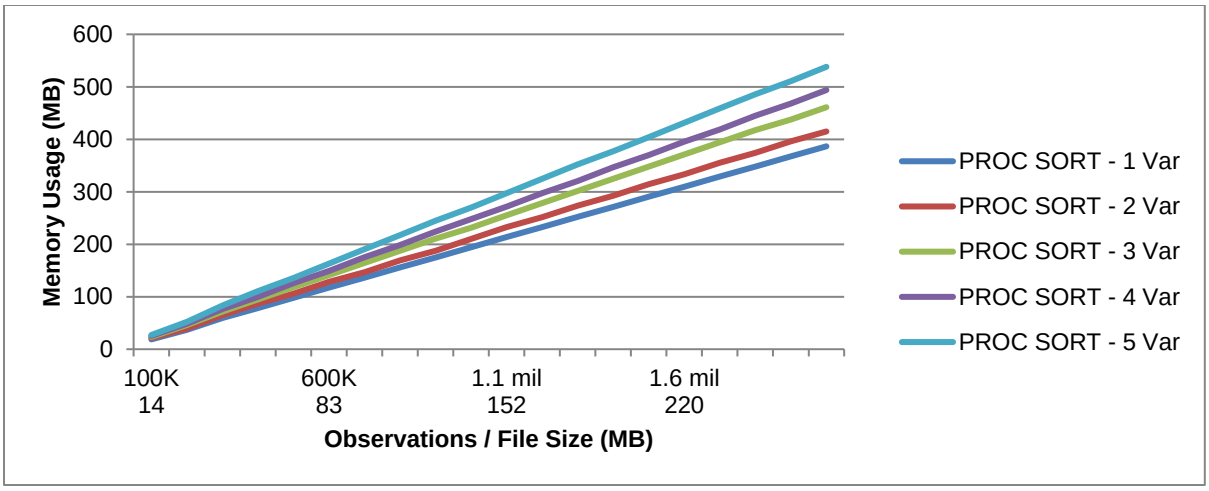

Figure 8. Memory Usage of SORT Procedure (Sorting by One to Five Character Variables)

- **Compression of Data** – File compression can be a preferred option for SAS environments that lack sufficient storage space, and for data sets that are referenced infrequently or intended to be archived. Compression requires additional processing and thus can incur longer execution time if the compression does not reduce the file size substantially. In cases where file size can be significantly minimized, compression can both reduce disk space usage and runtime, thus tremendously increasing the efficiency of sorting routines. Of note, the compression in no way affects memory usage.

  Because the sample data set includes no missing values and because each character variable always contains 20 characters, this data set is not a good candidate for compression. In fact, when compressed during the SORT procedure, the output data set actually *increases* in size by approximately 12 percent, from 275 MB to 308 MB. Notwithstanding the log note that describes this increase, the procedure still attempts to compress the data set, executing more slowly due to the actual increased file size.

```
proc sort data=original out=final (compress=char);
    by char1;
run;

NOTE: There were 2000000 observations read from the data set WORK.ORIGINAL.
NOTE: The data set WORK.FINAL has 2000000 observations and 10 variables.
NOTE: Compressing data set WORK.ORIGINAL increased size by 11.92 percent.
      Compressed is 4931 pages; un-compressed would require 4406 pages.
NOTE: PROCEDURE SORT used (Total process time):
      real time           14.75 seconds
      cpu time            6.09 seconds
```

  Although compression does sometimes increase file size, as demonstrated in this output, developers often can predict data sets that instead will benefit from compression. For example, data sets with substantial missing data or unused character space are prime contenders for beneficial compression. To illustrate this, the structure of the sample data set is modified so that the five character variables are 50 characters in length rather than 20, while the values inside those variables remain 20 characters in length. Thus, each new character variable has 30 characters of white space that can be compressed. At 200 million observations, this new format increases the uncompressed file size from 275 MB to 566 MB, although the new data set can be compressed to 333 MB, as demonstrated in the following output.

```
proc sort data=original out=final (compress=char);
    by char1;
run;
```

```
NOTE: There were 2000000 observations read from the data set WORK.ORIGINAL.
NOTE: The data set WORK.FINAL has 2000000 observations and 10 variables.
NOTE: Compressing data set WORK.ORIGINAL decreased size by 41.15 percent.
      Compressed is 5326 pages; un-compressed would require 9050 pages.
NOTE: PROCEDURE SORT used (Total process time):
      real time           3:00.01
      cpu time            23.99 seconds
```

Despite the significantly decreased file size after compression (by 41 percent), sorting runtime still was not significantly decreased with compression, even when the compression occurred in a DATA step before the SORT procedure, as depicted in Figure 9. This comparison involves three sorting algorithms, the first a traditional sort of an uncompressed data set, the second a sort of a compressed data set into a new data set (which still requires the COMPRESS statement in the SORT procedure), and the third a sort of a compressed data set in situ (i.e., no OUT statement, which requires no COMPRESS statement). Thus, for both compression algorithms, execution time was more closely correlated with the file size of the larger, uncompressed file than the compressed one. This illustrates that despite the significantly decreased file size (and corresponding decrease in input/output (I/O) resources required to write the smaller output file), the I/O resources required for handling the compressed data set eliminate any runtime efficiency gains. Notwithstanding, this compression still demonstrates much more efficient data storage and a very significant reduction in file size that could incredibly benefit resource-constrained teams while at no cost to runtime performance.

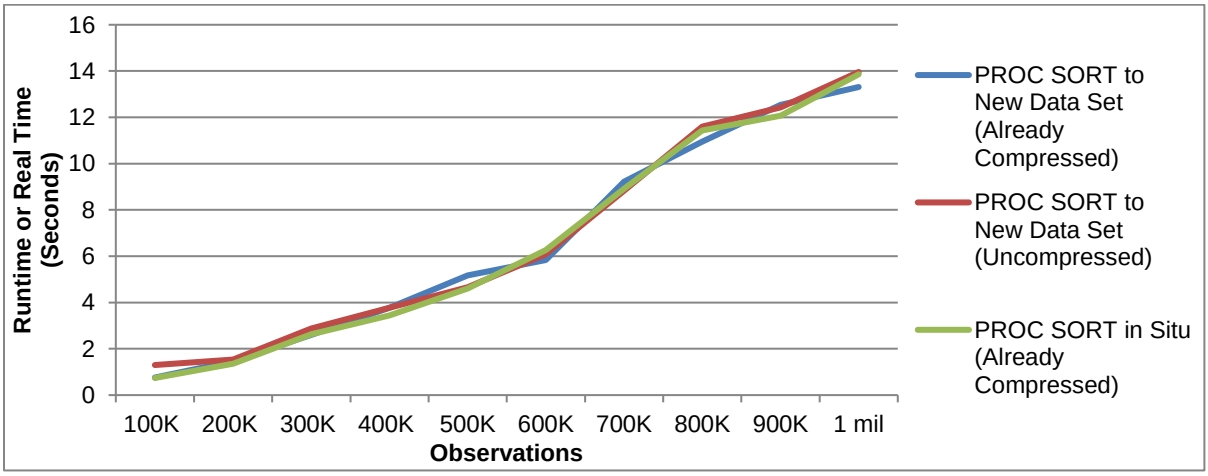

Figure 9. Runtime for Sorting Uncompressed and Compressed Data Sets by One Numerical Variable

By modifying the sample data set further by lengthening each character variable to 100 characters (yet maintaining values that are only 20 characters in length), an even greater compression ratio can be achieved. The following output demonstrates achieving a 68 percent compression ratio when 1 million observations are sorted with the SORT procedure, the performance metrics of which are depicted in Figure 10.

```
proc sort data=original out=final (compress=char);
    by num1;
run;

NOTE: There were 1000000 observations read from the data set WORK.ORIGINAL.
NOTE: The data set WORK.FINAL has 1000000 observations and 10 variables.
NOTE: Compressing data set WORK.FINAL decreased size by 68.01 percent.
      Compressed is 2666 pages; un-compressed would require 8334 pages.
NOTE: PROCEDURE SORT used (Total process time):
      real time           47.94 seconds
      cpu time            26.93 seconds
```

With this tremendous 68 percent reduction in file size, the compression I/O trade-off finally pays off with both increased storage efficiency *and* increased runtime efficiency. Figure 10 demonstrates that with significant compression ratios, the sorting performance will faster on compressed than uncompressed data sets. The two most efficient methods involve sorting a compressed data set in situ (in which no OUT statement is used) and sorting a compressed data to a new data set (which requires the COMPRESSION option). The COMPRESSION option is not necessary when sorting a compressed data set in situ because the data set name does not change so its compression characteristics also remain static. The next most efficient method involves sorting uncompressed data into a new data set, which requires the COMPRESSION option. Because the data are not previously compressed, this requires added I/O resources and runtime to complete. The slowest method involves sorting an uncompressed data set without compression, which requires significantly longer runtime due to the larger file that must be written. In all cases, memory usage was equivalent between these four methods.

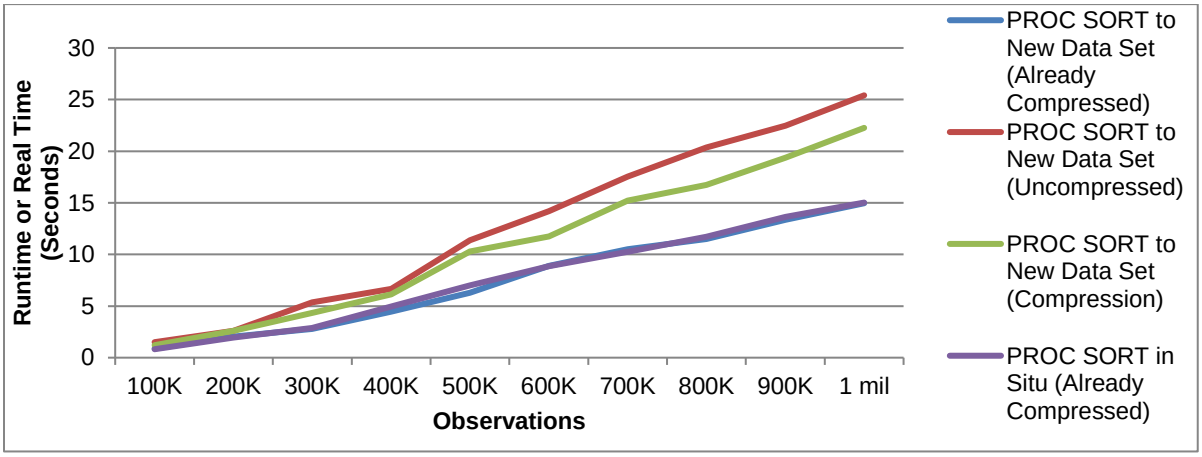

Figure 10. Runtime for Sorting Uncompressed and Compressed Data Sets by One Numerical Variable

- **TAGSORT Option** – The TAGSORT option in the SORT procedure facilitates sorting larger data sets in environments having constrained memory. As demonstrated in Figure 11, when sorting between 100,000 and 1 million observations of the sample data set, the TAGSORT option eclipses all other sorting methods in terms of memory efficiency, requiring substantially reduced memory to complete.

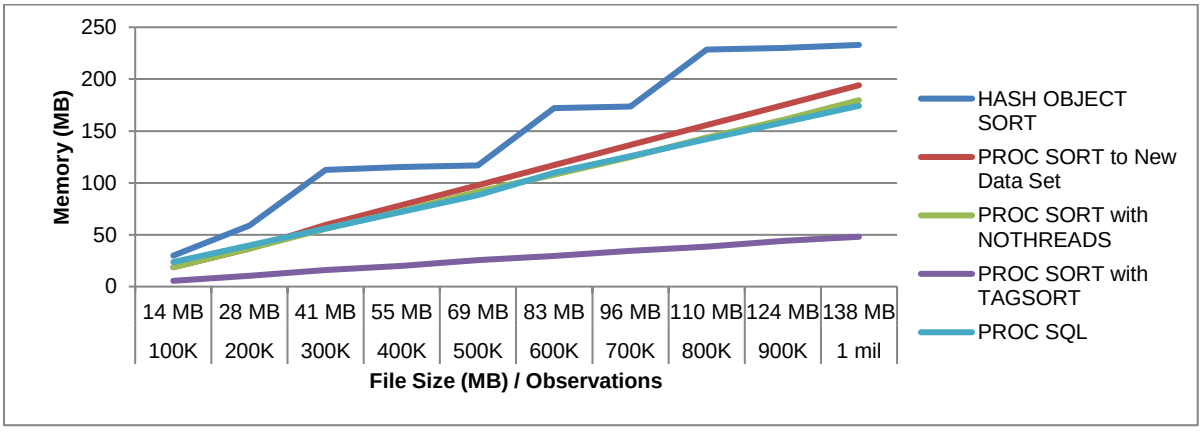

Figure 11. Decreased Memory Utilization with SORT Procedure and TAGSORT Option

The TAGSORT option unfortunately requires exponentially longer runtime due to increased I/O processing that compensates for the reduction in memory usage. As Figure 12 demonstrates, this increased runtime is so dramatic that the TAGSORT option should only be considered in environments with extreme resource limitations.

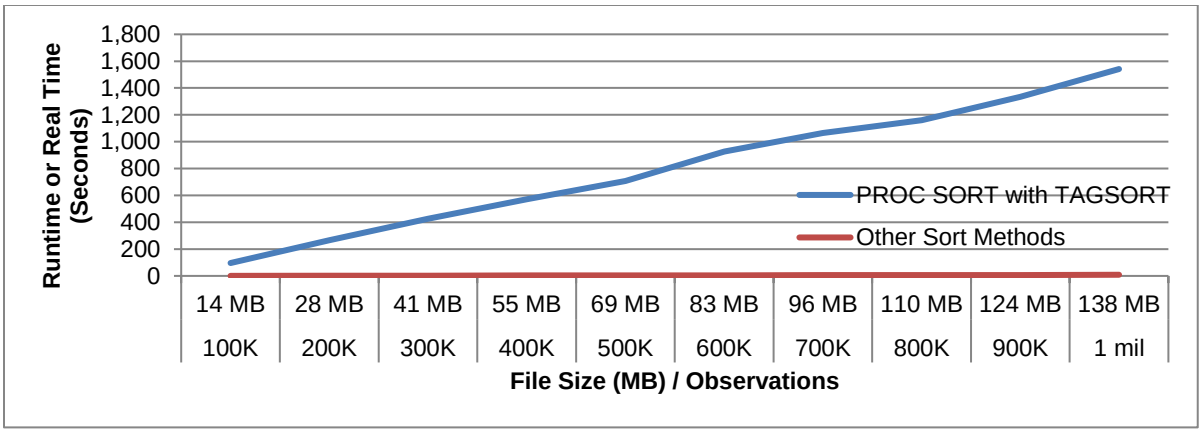

Figure 12. Substantially Increased Runtime with SORT Procedure and TAGSORT Option

- **SAS Interface, Environment, and Hardware** – The performance achieved from the SAS Display Manager on a single laptop may be unrecognizable in comparison to performance achieved on a multi-processor SAS grid environment. The SAS software interface and version, inasmuch as its OS, network connection, and hardware each can influence performance and resource utilization. Because many of these attributes are immutable or rarely change in a given environment, their influences may go unrecognized to SAS practitioners who work on a single SAS platform. In fact, not until working across various platforms will some practitioners recognize the substantial impact that nonprogrammatic factors play on sorting performance and efficiency.

  Consider the researcher who has utilized the free version of SAS Studio provided generously from SAS through the SAS University Edition. He's accustomed to performance of the SORT procedure akin to Figure 4, which was generated on that software, and might mistakenly believe that purchasing the SAS Display Manager would yield equivalent results. However, when run on the same machine, both the SAS Display Manager 9.4 and Enterprise Guide 7.1 significantly outperform the SAS University Edition, even when the CPUCOUNT is reduced to two to account for the CPUCOUNT=2 restriction imposed by the SAS University Edition, and the SORTSIZE is kept constant at 1 GB. Figure 13 demonstrates the sample data set sorted by one numeric variable with the SORT procedure and iterated from 100,000 to 4 million observations.

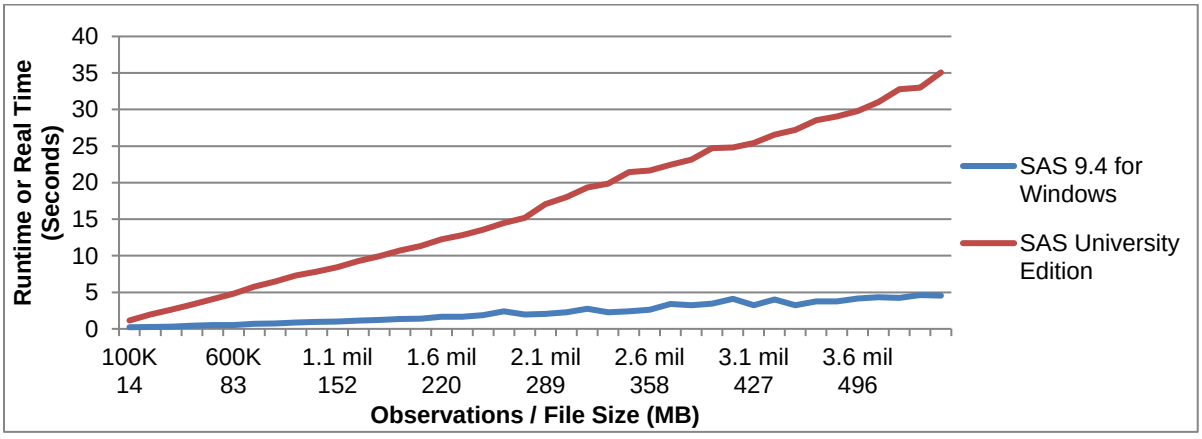

Figure 13. SORT Procedure Efficiency Compared Between SAS Software Applications

At 4 million observations, the SORT procedure is still performing efficiently in each respective software platform, as demonstrated by the two functions which remain relatively linear. However, because at 4 million observations the SAS Display Manager completes in 4.5 seconds, it can be said to be significantly more efficient than the SAS University Edition, which completes in 35.1 seconds, almost eight times slower. Moreover, by expanding the samples to 4.5 million observations, the *inefficiency elbow* of the SORT procedure in the SAS University Edition

can be observed, while SAS Display Manager continues to function efficiently. In fact, as demonstrated in Figure 14, at 4.5 million observations, SAS Display Manager completes the same SORT procedure in 6.0 seconds while the SAS University Edition completes in four and one half hours, over 2,600 times slower! Of note, each SAS interface used identical memory while demonstrating these significantly divergent performance results.

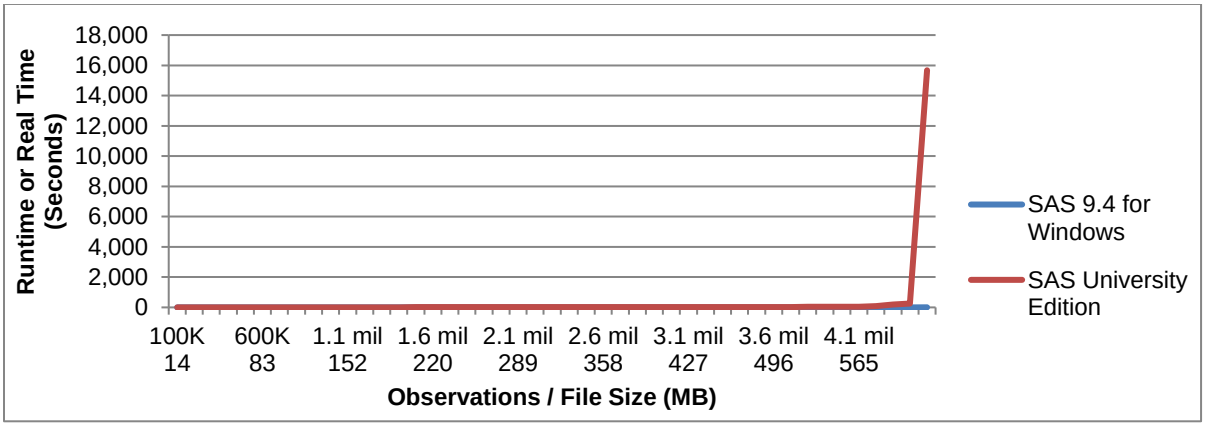

Figure 14. SORT Procedure Efficiency Compared Between SAS Software Applications

At some larger file size, the SAS Display Manager will also experience a similar inefficiency elbow to that of the SAS University Edition in Figure 14. Thus, regardless of implementation, at some point every sort will become slow and, at some later point, will fail outright. This highlights the critical role that the SAS interface, hardware, infrastructure, and network can play on sorting performance. It also demonstrates that because greater sorting performance can be achieved through both programmatic and nonprogrammatic solutions, in some organizations it may be cheaper or more efficient to purchase additional software, hardware, or infrastructure than to invest the time and resources into developing a programmatic solution. It conversely demonstrates that even when provided the latest and most powerful software, hardware, and infrastructure, sorting can be made slow or inefficient when implemented through programmatic solutions that are less than ideal. Thus, only by combining both programmatic and nonprogrammatic solutions can sorting efficiency truly be maximized.

FACTORS THAT CAUSE SORTING FAILURE

While sorting failure can be caused by syntax errors, such as misspelling the data set name or a sort variable, a more common failure source in production environments is resource limitations. Sorting will fail when all available memory or disk space is used but efficiency losses are often observed long before outright failure. Sorting algorithms often begin to perform extremely inefficiently when available random access memory (RAM) is depleted and must be substituted with virtual memory. This section depicts failed sorting processes that cause runtime errors. In later sections, demonstrations include not only how to avoid failed sorting routines but also how to bypass routines that are likely to operate inefficiently due to data set size or other data set or environmental attributes.

- **Disk Volume Full** – Finite disk space will cause sort failures when either the entire disk volume is full or the WORK library is full. Typically when the disk volume is full, it's time to start deleting and archiving files as well time to purchase new storage media. Because SAS requires temporary data sets (called utility files) to sort data, often much more space is required to perform the sort than to store the final, sorted data set. The following output demonstrates failure of the SORT procedure when the disk volume being written to becomes full.

```
proc sort data=safesort.original out=safesort.final;
    by char1;
run;

ERROR: No disk space is available for the write operation.  Filename =
       C:\Users\me\AppData\Local\Temp\SAS Temporary
       Files\SAS_util0001000003CC_comp\ut03CC000029.utl.
ERROR: Failure while attempting to write page 1352 of sorted run 15.
```

```
ERROR: Failure while attempting to write page 44934 to utility file 1.
ERROR: Failure encountered while creating initial set of sorted runs.
ERROR: Failure encountered during external sort.
ERROR: Sort execution failure.
NOTE: The SAS System stopped processing this step because of errors.
NOTE: There were 80493346 observations read from the data set SAFESORT.ORIGINAL.
WARNING: The data set SAFESORT.FINAL may be incomplete.  When this step was stopped
    there were 0 observations and 10 variables.
NOTE: PROCEDURE SORT used (Total process time):
      real time           9:55.38
      cpu time            2:15.97
```

While this error occurs while the SORT procedure is attempting to write utility files, the SORT procedure can also fail if the disk runs out of space while trying to combine these utility files into a single, sorted data set.

```
proc sort data=safesort.original out=safesort.final;
    by char1;
run;

NOTE: There were 200000000 observations read from the data set SAFESORT.ORIGINAL.
ERROR: Insufficient space in file SAFESORT.FINAL.DATA.
ERROR: Failure while merging sorted runs from utility file 1 to final output.
ERROR: Failure encountered during external sort.
ERROR: File SAFESORT.FINAL.DATA is damaged. I/O processing did not complete.
NOTE: The SAS System stopped processing this step because of errors.
WARNING: The data set SAFESORT.FINAL may be incomplete.  When this step was stopped
    there were 180797314 observations and 10 variables.
NOTE: PROCEDURE SORT used (Total process time):
      real time           53:03.88
      cpu time            9:58.40
```

- **WORK Library Full** – Another runtime error type occurs when the WORK library runs out of disk space yet the disk volume itself is not full. The following code demonstrates process failure that occurs when a large data set is sorted into a data set located in the WORK library.

```
proc sort data=save.original out=final;
    by byvar;
run;
%put &syserr / &syserrortext;

NOTE: There were 3000000 observations read from the data set SAVE.ORIGINAL.
ERROR: Insufficient space in file WORK.FINAL.DATA.
ERROR: Sort client has encountered an error.
ERROR: Failure encountered during internal sort.
ERROR: File WORK.FINAL.DATA is damaged. I/O processing did not complete.
NOTE: The SAS System stopped processing this step because of errors.
WARNING: The data set WORK.FINAL may be incomplete.  When this step was stopped
there were 2158295 observations and 10 variables.
NOTE: PROCEDURE SORT used (Total process time):
      real time           18.41 seconds
      cpu time            6.17 seconds

1012 / File WORK.TEST.DATA is damaged. I/O processing did not complete.
```

In some cases, the WORK library may be full from previous processes that have created temporary data sets that are no longer needed. Deleting all contents of the WORK library with the DATASETS procedure can often allow subsequent sort processes to succeed.

```
proc datasets library=work kill nolist;
run;
quit;
```

If the WORK library still cannot accommodate the sorted output data set, it can be sorted into a named SAS library.

```
proc sort data=save.original out=save.final;
    by byvar;
run;
```

If the current disk volume is full, other volumes (if available) can be specified to sort the data set to a location that has more space.[vi] The COMPRESSION option also can dramatically reduce disk space usage, and may also overcome space limitations that are causing disk space errors. Ultimately, however, when a programmatic solution cannot be implemented for disk space limitations, increased storage must be added.

- **Utility File Failure –** With default multithreaded processing enabled for the SORT procedure, SAS produces temporary data sets (utility files) that are ultimately merged into the final, sorted data set. By default, these utility files are saved to the location of the WORK library as specified in the UTILLOC system option, although they are not viewable within the SAS session.[vii] A low-memory environment can cause failure of these utility files, which can be triggered by substantially reducing the sorting memory with the SORTSIZE option. In the following example, the SORT procedure will fail regardless of whether the output data set is located in the WORK directory (as demonstrated) or in a named SAS library.

```
options sortsize=10;
proc sort data=original out=final;
    by byvar;
run;
%put &syserr / &syserrortext;

ERROR: Insufficient space in file WORK.'SASTMP-000000546'n.UTILITY.
ERROR: File WORK.'SASTMP-000000546'n.UTILITY is damaged. I/O processing did not
complete.
NOTE: The SAS System stopped processing this step because of errors.
NOTE: There were 1778160 observations read from the data set WORK.ORIGINAL.
WARNING: The data set WORK.FINAL may be incomplete.  When this step was stopped
there were 0 observations and 10 variables.
ERROR: Insufficient space in file WORK.FINAL.DATA.
NOTE: PROCEDURE SORT used (Total process time):
      real time           4.69 seconds
      cpu time            2.27 seconds

1012 / File WORK.'SASTMP-000000804'n.UTILITY is damaged. I/O processing did not
complete.
```

One remedy to utility file errors—so long as sufficient disk space and memory exist—is to modify the default UTILLOC location from the WORK library to another disk volume or location.

- **Memory Failure** – Strictly speaking, while this error is triggered indirectly by low-memory levels, the error results directly from the lack of disk space. Actual memory errors generate an error code value of 1016, as demonstrated in the following output.

```
ERROR: Hash object added 8912880 items when memory failure occurred.
```

```
FATAL: Insufficient memory to execute DATA step program. Aborted during the
EXECUTION phase.
ERROR: The SAS System stopped processing this step because of insufficient memory.
NOTE: DATA statement used (Total process time):
      real time           50.90 seconds
      cpu time       48.43 seconds

%put &SYSCC;
1016
```

The XMRLMEM system option displays (in bytes) the amount of memory allocated to SAS. The following code prints this value in MB.

```
%let memsize=%sysfunc(int(%sysevalf(%sysfunc(getoption(xmrlmem))/1048576)));
%put &memsize MB;
```

Total SAS memory, however, is not available for sorting routines, but rather is specified with the system option SORTSIZE that dictates how many bytes are available. The SORTSIZE option is defaulted to 1 GB and can be assessed (in megabytes) with the following code. [viii]

```
%let sortsize=%sysfunc(int(%sysevalf(%sysfunc(getoption(sortsize))/1048576)));
%put &sortsize;
```

Thus, to increase the SORTSIZE, the total memory may need to be increased first. In environments that have sufficient memory but which may not be utilizing it fully, increasing SORTSIZE may be the fastest way to facilitate sorts that are more reliable and robust to memory errors.

I. SINGLE-STEP SORTING METHODS

Although the focus of this text is not to contradistinguish SAS sorting methods, some explanation is required to describe how single-process sorting methods utilize resources. Because all divide-and-conquer sorting techniques—including the SAFESORT macro—rely on iterating through a single-process sort, divide-and-conquer techniques can improve performance if their underlying sorting routines also are efficient and carefully selected among alternatives. As demonstrated in the previous figures, Base SAS offers several methods and options to sort data.

1. **PROC SORT (in Situ) to Original Data Set** – The SORT procedure is the default sorting routine for many SAS practitioners and is most widely demonstrated throughout SAS literature. With the optional OUT statement omitted, the original data set is sorted in situ; however, undetected to the user, a temporary data set is created in the background so that if the SORT procedure fails, the original data set is not corrupted.

```
proc sort data=original;
    by byvar1 byvar2;
run;
```

2. **PROC SORT to New Data Set** – This SORT procedure differs from the previous technique only because a separate output data set is specified with the OUT statement. Performance typically is equivalent to sorting in situ.

```
proc sort data=original out=final;
    by byvar1 byvar2;
run;
```

3. **PROC SORT to New Data Set with TAGSORT Option** – The TAGSORT option is described more extensively in the *Base SAS 9.4 Procedures Guide* but operates by storing only sorting variables in a temporary data set, after which those "tags" are used to merge the remaining variables.[ix] Because only sorting variables are sorted, memory consumption can be considerably less than other sorting methods, but runtime almost always will be increased exponentially, substantially decreasing the value of this option.

```
proc sort data=original out=final TAGSORT;
    by byvar1 byvar2;
run;
```

4. **PROC SORT to New Data Set with NOTHREADS Option** – Threaded processes allow programs to multitask through parallel processing. For example, a common divide-and-conquer method for sorting first divides a data set into two (or more) subsets, after which these subsets can be sorted and joined concurrently in parallel before they are joined at the end of the process. Threading became available for SORT and other procedures beginning in SAS 9, and David Shamlin provides a comprehensive introduction to SAS threads.[x] Because THREADS is the default SAS option, the NOTHREADS options is presented here solely to simulate performance that would have occurred in older (single-threaded) versions of SAS.

```
proc sort data=original out=final NOTHREADS;
    by byvar1 byvar2;
run;
```

5. **PROC SORT to New Data Set with COMPRESS Option** – The data set COMPRESS option can be specified with the OUT statement to compress the sorted data set, either through character or binary methods.[xi] Compressed data sets will require less disk space but often take longer to create due to the compression process. If the initial data set is already compressed, the output data set automatically will be compressed if no OUT statement is supplied and the data set is sorted in situ. File compression represents one of the primary programmatic vehicles to overcome disk space limitations. Thus, while COMPRESS technically does not represent a different sorting methodology, due to its impact on sorting performance, it is included for comparison.

```
proc sort data=original out=final (compress=CHAR);
    by byvar1 byvar2;
run;
```

6. **PROC SORT of Compressed Data to New Data Set** – As demonstrated in Figure 10, sorting a data set that was previously compressed will be faster than performing the compression in the sort. Sorting a compressed data set can be accomplished in situ or can produce a new data set, the latter method of which does require the COMPRESS option.

```
data original (compress=CHAR);
    set original_not_compressed;
run;

* IN SITU Method;
proc sort data=original; * compressed implicitly;
    by byvar1 byvar2;
run;

* NEW DATA SET Method;
proc sort data=original out=final (compress=CHAR); * compressed explicitly;
    by byvar1 byvar2;
run;
```

7. **PROC SQL with ORDER BY Statement** – A benefit of the SQL procedure is that multiple SQL statements can be grouped within a single SQL procedure, often allowing it to replace not only the SORT procedure but also one or more accompanying DATA steps. Additional information can be found in the *SAS 9.4 SQL Procedure User's Guide*.[xii] Because the focus of this text is on sorting alone, only the most basic SQL sort is demonstrated, which significantly underrepresents the potential of the SQL procedure to sort while also manipulating data.

```
proc sql noprint;
    create table final as
    select * from original
```

```
    order by byvar1, byvar2;
    quit;
run;
```

8. **HASH Object with ORDER BY Statement** – Hash objects utilize lookup keys to place large amounts of data in memory to perform in-memory sorts. The hash object can be more efficient with significant available memory but will fail when memory is exhausted and, as Figure 3 demonstrates, is extremely susceptible to inefficient performance in low-memory environments. Additional information about the hash object can be found in the *SAS 9.4 Language Reference*.[xiii]

```
data _null_;
    if 0 then set original;
    declare hash hashy (dataset:'original', ordered:'Y') ;
           hashy.definekey ('byvar1','byvar2');
           hashy.definedata ('byvar1','byvar2','othervar1','othervar2','othervar3');
           hashy.definedone ();
           hashy.output(dataset:'final');
    stop;
run;
```

ASSESSING EFFICIENCY AND RESOURCE UTILIZATION

Because runtime is a commonly used metric for program efficiency, it is often captured programmatically for later demonstration, analysis, and comparison against stated performance requirements. Checking the system clock with the DATETIME function immediately before and after a process captures its total runtime in seconds.

```
options nofullstimer;
%let dtgstart=%sysfunc(datetime());
proc sort data=original out=final;
    by byvar;
run;
%let dtgstop=%sysfunc(datetime());
%let seconds=%sysevalf(&dtgstop-&dtgstart); /* seconds to complete sort */
%put SECONDS: &seconds;
NOTE: There were 3300000 observations read from the data set SAFESORT.TOWER.
NOTE: The data set SAFESORT.FINAL has 3300000 observations and 10 variables.
NOTE: DATA statement used (Total process time):
      real time                          1:21.67
      cpu time                           32.61 seconds

SECONDS: 1.22635020256042
```

Another method to capture runtime—in addition to numerous other performance metrics—is the FULLSTIMER system option, which is described fully in the *SAS Scalability and Performance Knowledge Base*.[xiv]

```
options fullstimer;
proc sort data=original out=final;
    by byvar;
run;
NOTE: There were 3300000 observations read from the data set SAFESORT.TOWER.
NOTE: The data set SAFESORT.FINAL has 3300000 observations and 10 variables.
NOTE: DATA statement used (Total process time):
      real time                          1:21.67
      user cpu time                      32.61 seconds
      system cpu time                    3.88 seconds
      memory                             771786.95k
```

```
OS Memory                       797500.00k
Timestamp                       10/15/2015 02:27:38 AM
Step Count                      2987  Switch Count  43
Page Faults                     2937
Page Reclaims                   148348
Page Swaps                      0
Voluntary Context Switches      59839
Involuntary Context Switches    840
Block Input Operations          156720
Block Output Operations         8
```

The difficulty of using only execution time or runtime (referenced as "real time" in the FULLSTIMER log) as a performance metric is that it can be heavily influenced by other processes that may be running concurrently. Base SAS documentation distinguishes a second measure of performance, CPU time, stating "It is not advisable to use real time as the only criterion for the efficiency of your program. The reason is that you cannot always control the capacity and load demands on your system. A more accurate assessment of system performance is CPU time, which decreases more predictably as you modify your program to become more efficient.[xv]" Thus, a user on a closed SAS system (such as a single laptop installation of the SAS Display Manager that is unsusceptible to automatic updates, virus scans, and other resource drains) may be able to measure and alter performance reliably from runtime alone. However, users in a multi-user or networked environment susceptible to competing resource utilization demands from SAS and other applications should rely more heavily on CPU time as a performance metric during performance tuning.

Notwithstanding this admonition regarding runtime usage, the performance of SAS programs in production is ultimately judged by runtime because it is the real-world construct that users, developers, and other stakeholders will observe. For this reason, while CPU time should often be utilized in development and testing to optimize program performance, runtime will always be used in service level agreements (SLAs), project requirements documentation, and other baseline metrics against which the performance of software ultimately is judged. And, if runtime begins to increase substantially in a production environment, other FULLSTIMER metrics such as CPU time and Memory should be evaluated to help diagnose the cause.

II. MULTI-STEP SORTING METHODS

As noted throughout the previous examples, as data set file size increases or available resources decrease, sorting routines will begin to behave inefficiently and eventually fail. *Failure* often is distinguished as and used interchangeably with the incidence of runtime errors (e.g., 1012 or 1016) that occur when SAS code terminates abruptly. A broader definition of failure, however, also encompasses sorting processes that do complete but complete so slowly that they provide reduced or no business value. In Figure 4, the SORT procedure took 3.5 hours to sort just 5 million observations, producing a valid, sorted data set but with tremendous inefficiency. Had the 5 million observation sort been performed with the same efficiency and performance observed at 4.5 million observations, the SORT procedure instead would have completed in only a couple minutes. Thus, this substantial deviation can demonstrate performance failure even when errors are not present and illustrates the need to implement multi-step sorting methods not only to replace *terminated* sorting routines but also *inefficient* ones in which no error occurred.

The difficulties of sorting large data sets are discussed in many texts, although a programmatic solution is often not provided for software that continues to fail during execution.[xvi] Tips such as "delete unneeded variables" or "delete unneeded observations" may be bandied about, but the refined data set still may fail to sort or sort inefficiently even after these refinements. Divide-and-conquer techniques instead offer a programmatic solution that breaks large data sets into chunks and successively sorts those chunks—either in series or parallel—to produce a single, sorted data set. As discussed, the single-step SORT procedure uses a multithreaded (i.e., parallel) divide-and-conquer algorithm, which is faster than single-threaded processing. However, if the SORT procedure fails due to large file size, a multi-step approach can implement successive calls to the SORT procedure to sort smaller chunks of the data set in series. Thus, multi-step sorting methods may increase runtime but will be more robust to failure.

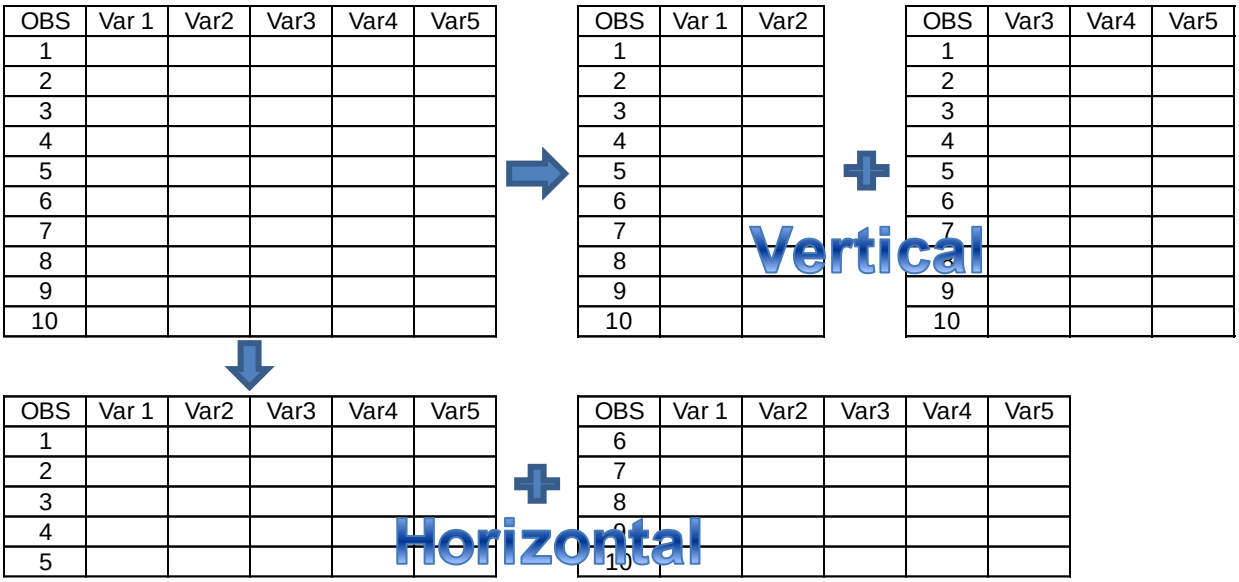

Figure 15. Demonstration of Horizontal and Vertical Data Subsets

The first step in a multithreaded sort is to subset the data set into smaller chunks. Two primary methods exist to subset unsorted data sets—subsetting vertically and horizontally, as depicted in Figure 15. In a vertical subset, some variables are eliminated from the data set after which this leaner data set is sorted, resulting in increased performance. The eliminated variables are later rejoined to the lean data set to produce the final sorted data set. This method effectively replicates how the TAGSORT option in the SORT procedure functions and can be effective on data sets that are very wide or which have extraordinarily long character fields that slow sorting routines but which are not critical to sorting order. Vertical subsetting is not described further in this text but should be considered in these instances.

Horizontal subsetting retains all variables and instead divides the data set into two or more chunks of observations. Within horizontal subsetting, two secondary methods exist by which the observations are parsed and divided—by *value* and by *position*. A horizontal subset *by value* establishes two or more value ranges that collectively cover all possible sort variable values, and which are used to separate chunks of observations. A horizontal subset *by position* divides observations sequentially by observation number, thus the first half of the data set might be output to one data set and the second half to a second data set. The SAFESORT macro utilizes a horizontal subset by position in its divide-and-conquer algorithm, but the history, strengths, and weaknesses of these two techniques first are briefly discussed.

HORIZONTAL SUBSET AND SORT BY VALUE

This method appears in earlier SAS literature but has fallen out of favor. One obvious disadvantage is that because value ranges must first be constructed to form the bounds of each data subset, the formats and ranges of all sorting variables must be known before the sort. Ranges also ideally should be constructed so that they equivalently distribute observations into similarly sized subset data sets, which moreover requires knowledge about the distribution of values. While a distribution table can be approximated by collecting a random sample of observations of sorting variables, this extra step delays processing.

A second major detraction to this method is that because conditional logic routines must be constructed that include variable values in their statements (as opposed to only the observation number as in subsetting by position), special characters that could appear in sort variables can produce unintended results or errors, especially when range value delimiters include these special characters. This hurdle can be overcome programmatically with SAS macro language but this added complexity can make already convoluted macro code even more error-prone. The major benefit of this method is that when data sets are subset by value not position, the resultant subsets are sorted both *within* and *among* data sets, thus the more efficient APPEND procedure can be used to join data sets in sequence.

```
data temp1 temp2;
    set original;
```

```
    if var1<="M" then output temp1;         * thus all values of var1 in temp1;
    else output temp2;         * will be less than all values of var1 in temp2;
run;
proc sort data=temp1;
    by var1;
run;

proc sort data=temp2;
    by var1;
run;

proc append base=temp1 data=temp2;
run;
```

Two historical examples of value subsetting are demonstrated and, while no attempt was made to automate these methods or build reusable processes, the techniques demonstrate how resource limitations could be overcome in an era when memory and disk storage were scarce and expensive!

- 1988: *Sorting Massive Data Sets*[xvii] - In this text, the original data set is subset manually with conditional logic statements, after which these subset data sets are sorted manually and finally joined.
- 1992: *Manipulating Large Highway and Crash Datasets*[xviii] - This text lists the divide-and-conquer method as a "last resort," demonstrating a manual implementation that first divides a data set into two subset data sets by the value of the year, then sorts these subsets in series, and finally joins them.

HORIZONTAL SUBSET AND SORT BY POSITION

Subsetting by position is significantly more efficient than subsetting by value because it can be accomplished by touching each observation only once. By implementing the FIRSTOBS and OBS statements within the SORT procedure, the SAS processor can skip directly to the chunk of observations currently being sorted, bypassing observations that precede or follow. A slightly less efficient method is also demonstrated in literature in which the subsets are created instead with a single DATA step (similar to the previous code) and later sorted in series. Because the latter method includes the gratuitous DATA step, efficiency is reduced. In both methods, however, the sorted subset data sets must ultimately be joined into a single, sorted data set. Because the observations are only sorted *within* subset data sets but not *among* subset data sets, the APPEND procedure cannot be utilized to join the data sets.

Another obvious advantage of subsetting by position is that no information about the sort variables, their formats, values, or distributions must be known. Instead, only the number of observations must be assessed (to establish the number of subsets) and likely the file size (to calculate how many observations should be included in each subset). Each of these metrics can be assessed through SAS code in a few seconds. A hardcoded demonstration of subsetting by position follows.

```
proc sort data=original (firstobs=1 obs=10000) out=temp1;
    by var1;
run;
proc sort data=original (firstobs=10001 obs=20000) out=temp2;
    by var1;
run;

data final;
    merge temp1 temp2;
    by var1;
run;
```

In a production environment, the previous code would be operationalized with and iterated by SAS macro language, which should dynamically assign all values for FIRSTOBS and OBS as well as the data set names. Moreover, the data

sets Temp1 and Temp2 should be written to a permanent SAS library so that they do not risk exceeding the disk space threshold of the WORK library. This dynamism is demonstrated in the SAFESORT macro in Appendix A. The following texts demonstrate subsetting by position to sort large data sets.

- 1997: *A Method for Sorting a Large Data Set with Limited Memory*[xix] – This single-page text utilizes FIRSTOBS and OBS statements in the SORT procedure to subset and sort a large data set. The SAS macro language iterates through the SORT procedure to produce sorted subset data sets that are later joined.

- 1999: *So You're Running Out of Sort Work Space…Reducing the Resources Used by PROC SORT*[xx] – This text also uses the FIRSTOBS and OBS statements in the SORT procedure to subset and sort a large data set. Another advantage is that the code dynamically decides whether to execute the multi-step method based on the number of observations. A downside in this dynamic process, however, is that observation count alone (rather than the more highly predictive file size) is used as the threshold to invoke the multi-step sort.

- 2012: *Sorting a Large Data Set When Space is Limited*[xxi] – This text gives two possible methodologies to sort large data sets, the second of which uses the divide-and-conquer technique. The SPLITSORT macro is demonstrated but unfortunately uses a gratuitous DATA step to subset the data into chunks before these are sorted via successive SORT procedures. The conditional logic inside the DATA step additionally uses an inefficient IF-IF-IF technique rather than IF-THEN-ELSE, so unfortunately the method is comparatively inefficient.

THE SAFESORT SOLUTION

The SAFESORT macro builds upon these historical techniques with the following advancements to create a much more robust, reliable, reusable, and generalizable solution.

- A priori exception handling detects conditions (such as large file size) that predict sort inefficiency and failure. If triggered, process flow dynamically routes from a single-step to a multi-step sorting method.

- Post hoc exception handling detects failed single-step sort processes that have terminated with errors. If triggered, process flow dynamically routes from a single-step to a multi-step sorting method without interruption. When single-step sorting terminates with an error within the SAFESORT macro, this is often an indication that the &THRESH parameter should be lowered, thus after this error is detected the &THRESH value is automatically divided in half before the multi-step process is initiated. Although this improves robustness of the macro, developers should thereafter manually lower the &THRESH parameter to reduce the future likelihood of processing data sets that are too large for single-step sorting methods.

- Post hoc exception handling also verifies that the multi-step process subsequently completed without error through two global macro variable return codes, &SORTERR and &SAFESORTERR. If the multi-step sorting process also fails due to memory or other errors, this can signal that &THRESH should be reduced. At some point, however, even SAFESORT will fail due to resource limitations.

- Any of three sorting methods can be specified at invocation for both the preferred single-step and the secondary multi-step solution. Thus, SAS users can specify to sort data using SQL (when specified in the METH1 parameter) but, if that process fails or the data set is larger than the file size threshold, the data will be sorted with a multi-step process using PROC SORT (when specified in the METH2 parameter).

The SAFESORT macro is invoked with the following parameters:

```
%macro SAFESORT (meth1 = /* primary sorting algorithm (SORT, SQL, HASH) */,
    meth2 = /* secondary sorting algorithm (SORT, SQL, HASH) */,
    dsn = /* input data set name in LIB.DSN format */,
    dsnout = /* output data set name in LIB.DSN format */,
    sortvars = /* space-delimited list of sort variables */,
    thresh = /* file size (in MB) to implement multi-step sort */);
```

SAFESORT EXCEPTION HANDLING

To achieve maximum reusability, a dynamic sorting method must accommodate data sets of all shapes and sizes. When SAFESORT is invoked on a "normal-sized" data set that does not challenge resource limitations, a single-step sorting method should be executed rather than a multi-step method, as demonstrated in the following business logic.

```
%if %sysevalf(&filesize/1048576<&thresh) %then %do;
    %&meth1(dsn=&lib..&dsn, dsnout=&libout..&dsnout, obs1=1, obs2=&obstot,
          sortvars=&sortvars);
    %end;
```

This a priori exception handling can save hours of potentially wasted runtime by preventing sorting processes from executing that will be slow, inefficient, or fail with errors. The previous default business rules can be improved and made even more accurate by incorporating additional attributes that influence sorting efficiency such as number or format of sorting variables.

Post hoc exception handling plays an equally important role in the SAFESORT macro as it validates success of sorting routines. For example, if a warning or error occurs in the SORT, SQL, or HASH macros (which are called from inside the SAFESORT macro), the value of the global macro variable &SORTERR is set equal to the value of &SYSCC. Thus, if a single-step sort initially was directed and executed (because the file size was below the parameterized threshold) but still terminated with an error, SAFESORT automatically would transfer process flow to the multi-step sort method specified in the macro parameter &METH2. Post hoc exception handling also examines the value of &SYSCC after completion of every multi-step sort to validate success and, if a warning or error is detected, the value of &SYSCC is saved in the global macro variable &SAFESORTERR. In production SAS software in which processes are dependent on a prerequisite data sort, implementing the SAFESORT macro and subsequently checking its return codes can be used to demonstrate that the required sort completed without error.

INCREASED PERFORMANCE WITH SAFESORT

To implement SAFESORT most efficiently, system performance limitations as well as specific data set attributes should be understood in conjunction with each other. For example, Figure 16 demonstrates performance of the SORT procedure incrementing from 100,000 to 4 million observations across sample data sets of increasing size. Around 3.7 million observations, memory limitations begin to cause the procedure to slow as it crawls toward failure. The following SAFESORT invocation is demonstrated in Figure 16, which highlights the eventual performance benefits of SAFESORT over the traditional SORT procedure as the 4 million observation threshold is reached.

```
%macro SAFESORT (meth1=SORT, meth2=SORT, dsn=SAFESORT.original,
    dsnout=SAFESORT.final, sortvars=char1, thresh=100);
```

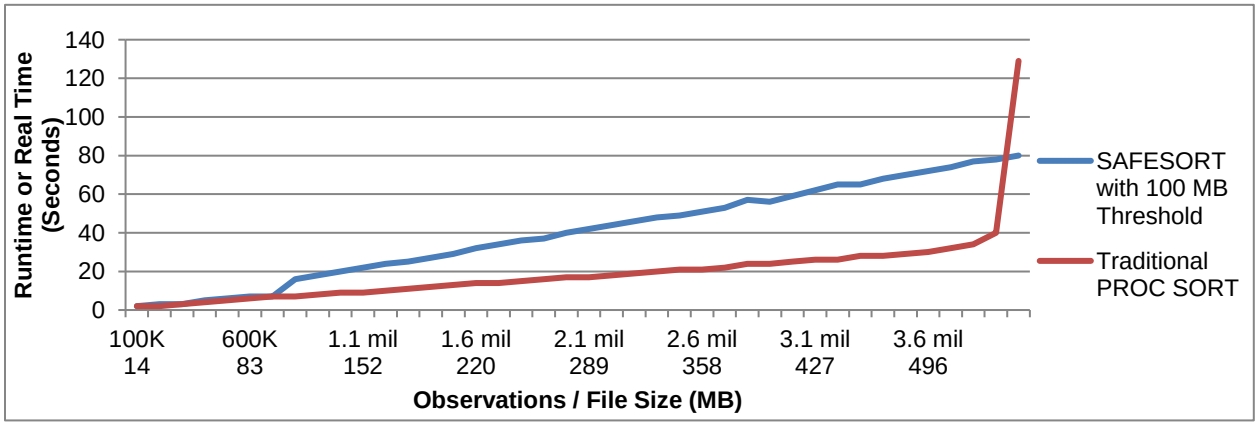

Figure 16. SAFESORT Improved Performance over Traditional SORT Procedure

With the SAFESORT threshold set at 100 MB, the macro performs the default, single-step sort method specified in the &METH1 parameter (i.e., the SORT procedure) on all data sets below 100 MB. This explains the identical performance achieved from both methods in file sizes ranging from 14 to 96 MB. However, as the file size crosses the 100 MB

threshold, the SAFESORT macro directs execution of the multi-step sort process, specified in the &METH2 parameter. Because the value of &METH2 is also SORT, the SORT procedure will also be utilized on large data sets but only after breaking them into smaller data subsets less than 100 MB.

After the 100 MB threshold is crossed, the divergence of the SAFESORT performance and the traditional SORT procedure is apparent, as SAFESORT requires additional runtime both to sort the data set into smaller chunks and finally to join these into a sorted output data set. The traditional SORT procedure consistently outperforms the multi-step SORT procedure so long as memory is plentiful. For example, at 3 million observations, SAFESORT multi-step SORT required 59 seconds while the traditional SORT procedure required only 25 seconds. But, at 4 million observations, after the SAS SORT procedure has begun to perform inefficiently, SAFESORT completes in 80 seconds while the SORT procedure completes in 2 minutes 9 seconds. Moreover, at 4 million observations the SORT procedure is using 800 MB of memory while SAFESORT uses only 166 MB of memory, roughly maintaining the level of memory consumption shown when sorting a 100 MB data set. In fact, the only increase in memory consumption of SAFESORT multi-step processes occurs in its final DATA step that merges the temporary data sets into the final, sorted product.

Because the SORT procedure was still performing efficiently on 100 MB data sets, in reality the &THRESH parameter should have been set higher, at some point immediately before the acceleration of the performance slope toward inefficiency. The following code demonstrates setting &THRESH to 500 MB, with performance demonstrated in Figure 17. SAFESORT with a 500 MB threshold now performs equivalently to the SORT procedure for files smaller than 500 MB and, as that threshold is crossed, runtime increases dramatically due to the shift from a single-step to multi-step sorting method. But, because fewer temporary 500 MB data sets than 100 MB data sets need to be joined in the final DATA step, the 500 MB threshold outperforms the 100 MB threshold. And, by 4 million observations, both SAFESORT options are significantly outperforming the traditional SORT procedure.

```
%macro SAFESORT (meth1=SORT, meth2=SORT, dsn=SAFESORT.original,
    dsnout=SAFESORT.final, sortvars=char1, thresh=500);
```

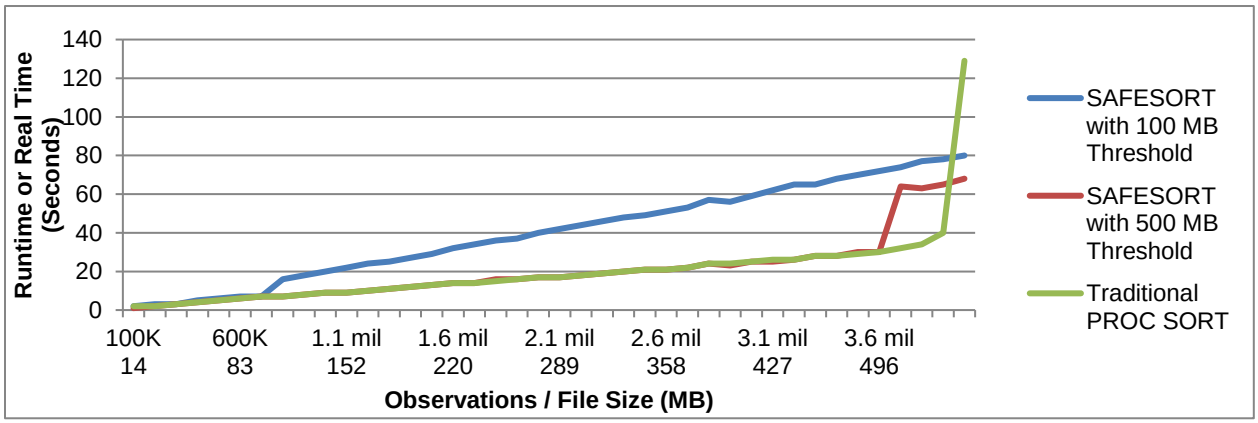

Figure 17. Necessity to Optimize SAFESORT &THRESH Parameter to Maximize Performance

In a final demonstration of the tremendous performance increase of SAFESORT over single-step methods, Figure 17 demonstrates runtime of SAFESORT compared to the traditional SORT procedure, using a 500 MB threshold and iterated from 100,000 to 4.5 million observations. At 4.5 million observations, the SORT procedure requires 4 hours 20 minutes to complete while SAFESORT finishes in only 74 seconds, over 200 times faster!

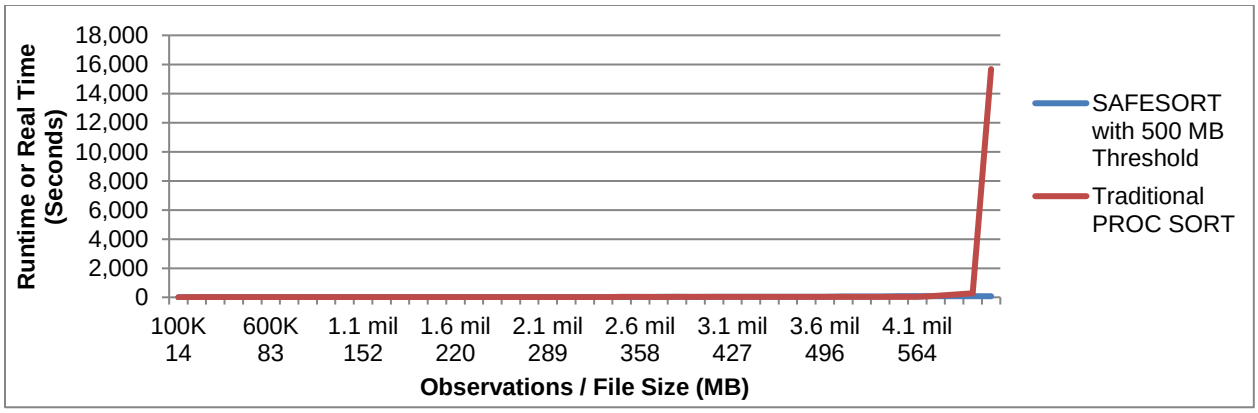

Figure 17. SAFESORT Dramatically Improved Performance over Traditional SORT Procedure

NEXT STEPS

Because this text focuses on a sample data set of standardized form and only variable length, the SAFESORT &THRESH parameter of roughly 500 MB could be assessed easily through trial and error. Implementation of SAFESORT in a production environment will first require assessment of the performance of single-step sorting processes to determine the inefficiency elbow after which sorting becomes inefficient. SAS practitioners might find, for example, that an extract-transform-load (ETL) process that runs daily includes one or multiple sorts that are performing past the inefficiency elbow, thus an ETL process could be made faster and more efficient with SAFESORT. Incorporating additional parameters into the SAFESORT invocation—such as number of variables, type of variables, number and type of sorting variables, or representation of missing data—would make the macro even more accurate, efficient, and generalizable.

While sorting speed is paramount, reliability and robustness are the primary of objectives of the SAFESORT macro. Over time, SAS practitioners can develop an intuitive sense of what size or type of data causes traditional sorting processes to fail in their specific environment. An optional modification to the SAFESORT macro could record all invocations of SAFESORT in a control table and could include FULLSTIMER and other performance metrics, demonstrating whether the single-step sorting process succeeded or failed. Manual analysis of this control table would assist developers in optimizing SAFESORT implementation for specific data sets, while automated processes could be constructed that would query the control table and, based on past performance, dynamically alter SAFESORT invocation parameters to optimize multi-step sorting.

A final advancement of SAFESORT would be its implementation to support parallel processing. While SAFESORT does use the divide-and-conquer technique to break the unsorted data set into manageable chunks, these are subsequently sorted in series rather than parallel. This creates a false dependency (e.g., because observations one to 1 million must be sorted before later observations) that delays processing, whereas ideally these chunks could be sorted concurrently and joined thereafter. In theory, if these chunks could be sorted concurrently, a parallel divide-and-conquer SAFESORT could execute. This concurrency would also overcome the unfortunate Base SAS limitation that increasing CPUCOUNT beyond a value of two seems to provide no further performance advantage when sorting.

CONCLUSION

The SAFESORT macro advances presently published big data sorting methodologies by improving reliability and robustness. The exception handling framework diverts to multi-step sorting automatically when single-step routines fail with warnings or runtime errors. This exception handling ensures that if really big data or very limited resources are encountered, return codes are generated that can alert subsequent, dependent processes. While the intent of SAFESORT is to prevent process failure, a concomitant benefit is the increased performance that can be gained by implementing SAFESORT on large data sets that are being sorted inefficiently with traditional, single-step methods. With an understanding of FULLSTIMER performance metrics from sort routines, SAS practitioners are well-placed to identify both the cause and effect of inefficient processes as well as to apply the SAFESORT solution to optimize future sorting.

REFERENCES

[i] Smith, Curtis A. *Programming Tricks for Reducing Storage and Work Space*. SAS Users Group International (SUGI) 27.

[ii] Jain, Wishal. 2010. *Working Efficiently with Large SAS Datasets*. Pharmaceutical Users Software Exchange (PhUSE).

[iii] SAS Press. *Scalability and Performance Community*. Retrieved from http://support.sas.com/rnd/scalability/index.html.

[iv] SAS Press. *SAS® 9.4 System Options: Reference, Fourth Edition. THREADS System Option*. Retrieved from https://support.sas.com/documentation/cdl/en/lesysoptsref/68023/HTML/default/viewer.htm#p0cvg7xpfvfyn4n1rnp64gu91poh.htm.

[v] SAS Press. *SAS® 9.4 System Options: Reference, Fourth Edition. CPUCOUNT= System Option*. Retrieved from https://support.sas.com/documentation/cdl/en/lesysoptsref/68023/HTML/default/viewer.htm#p14arc7flhenwqn1v1gipt9e49om.htm.

[vi] SAS Press. *SAS® 9.4 Companion for Windows, Fourth Edition. Advanced Performance Tuning Methods*. Retrieved from http://support.sas.com/documentation/cdl/en/hostwin/67962/HTML/default/viewer.htm#n0ea63jfjic0vpn15dbdh2ee0xbb.htm.

[vii] SAS Press. *SAS® 9.4 System Options: Reference, Fourth Edition. UTILLOC= System Option*. Retrieved from https://support.sas.com/documentation/cdl/en/lesysoptsref/68023/HTML/default/viewer.htm#p1texr4rxo0ipyn1ovajj11raccx.htm.

[viii] SAS Press. *SAS® 9.4 Companion for Windows, Fourth Edition. SORTSIZE System Option: Windows*. Retrieved from http://support.sas.com/documentation/cdl/en/hostwin/67962/HTML/default/viewer.htm#p024jq3a5zotf3n19rt86580m0cv.htm.

[ix] SAS Press. *SAS® 9.4 Procedures Guide, Fourth Edition. SORT Procedure. TAGSORT*. Retrieved from https://support.sas.com/documentation/cdl/en/proc/67916/HTML/default/viewer.htm#p02bhn81rn4u64n1b6l00ftdnxge.htm.

[x] Shamlin, David. 2004. *Threads Unraveled: A Parallel Processing Primer*. SAS Users Group International (SUGI) 29.

[xi] SAS Press. SAS® 9.4 System Options: Reference, Fourth Edition. *COMPRESS= System Option*. Retrieved from https://support.sas.com/documentation/cdl/en/lesysoptsref/68023/HTML/default/viewer.htm#n0uhpz2l79vy0qn12x3ifcvjzsyt.htm.

[xii] SAS Press. *SAS® SQL Procedure User's Guide, Second Edition. Sorting Data*. Retrieved from http://support.sas.com/documentation/cdl/en/sqlproc/68053/HTML/default/viewer.htm#p1jr2es0aus2qtn11peo724tt8f4.htm.

[xiii] SAS Press. *SAS® 9.4 Language Reference: Concepts, Fifth Edition. Using the Hash Object*. Retrieved from http://support.sas.com/documentation/cdl/en/lrcon/68089/HTML/default/viewer.htm#n1b4cbtmb049xtn1vh9x4waiioz4.htm.

[xiv] SAS Press. *Knowledge Base: FULLSTIMER SAS Option*. Retrieved from http://support.sas.com/rnd/scalability/tools/fullstim/index.html.

[xv] SAS Press. *SAS® 9.4 Language Reference: Concepts, Fifth Edition. Collecting and Interpreting Performance Statistics*. Retrieved from

http://support.sas.com/documentation/cdl/en/lrcon/68089/HTML/default/viewer.htm#n014p8dj7c4lqon1ngsytowdqgar.htm.

[xvi] Dabhi, Dhiraj. 2014. *Big Data – Step towards High Performance Analytics*. Pharmaceutical Users Software Exchange (PhUSE).

[xvii] Rubin, David S. 1988. *Sorting Massive Data Sets*. SAS Users Group International (SUGI).

[xviii] Hamilton, Elizabeth G. 1992. *Manipulating Large Highway and Crash Datasets*. Northeast SAS Users Group (NESUG).

[xix] LaBrecque, Leslie. 1997. *A Method for Sorting a Large Data Set with Limited Memory*. Northeast SAS Users Group (NESUG).

[xx] Virgile, Bob. 1999. *So You're Running Out of Sort Work Space…Reducing the Resources Used by PROC SORT*. SAS Users Group International (SUGI).

[xxi] Sridharma, Selvaratnam. 2012. *Sorting a Large Data Set When Space is Limited*. Northeast SAS Users Group (NESUG).

CONTACT INFORMATION

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

Name: Troy Martin Hughes
E-mail: troymartinhughes@gmail.com

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.

APPENDIX A. SAFESORT MACRO

```
* used by the SAFESORT macro to initiate PROC SORT;
%macro sort(dsn= /* data set being sorted */,
    dsnout= /* final data set name */,
    obs1= /* FIRSTOBS value */,
    obs2= /* OBS value */,
    sortvars= /* space-delimited list of BY variables */);
%global sorterr;
%let sorterr=;
%let syscc=0;
proc sort data=&dsn (firstobs=&obs1 obs=&obs2) out=&dsnout;
    by &sortvars;
run;
%if %eval(&syscc>0) %then %let sorterr=&syscc;
%mend;

* used by the SAFESORT macro to initiate PROC SQL sort;
%macro sql(dsn= /* data set being sorted */,
    dsnout= /* final data set name */,
    obs1= /* FIRSTOBS value */,
    obs2= /* OBS value */,
    sortvars= /* space-delimited list of BY variables */);
%let syscc=0;
%global sorterr;
%let sorterr=;
%let totvars=1;
%let byvars_comma=;
%do %while(%length(%scan(&sortvars,&totvars,,S))>1);
    %if &totvars>1 %then %let byvars_comma=&byvars_comma,
           %scan(&sortvars,&totvars,,S);
    %else %let byvars_comma=&byvars_comma %scan(&sortvars,&totvars,,S);
    %let totvars=%eval(&totvars+1);
    %end;
proc sql noprint;
    create table &dsnout AS
           select * from &dsn (firstobs=&obs1 obs=&obs2)
           order by &byvars_comma;
    quit;
run;
%if %eval(&syscc>0) %then %let sorterr=&syscc;
%mend;

* used by the SAFESORT macro to initiate hash object sort;
%macro hash(dsn= /* data set being sorted */,
    dsnout= /* final data set name */,
    obs1= /* FIRSTOBS value */,
    obs2= /* OBS value */,
    sortvars= /* space-delimited list of BY variables */);
%let syscc=0;
%global sorterr;
%let sorterr=;
%local i;
```

```
%let dsid=%sysfunc(open(&dsn, i));
%let nvars=%sysfunc(attrn(&dsid, nvars));
%let listvars=;
%do i=1 %to &nvars;
    %if %eval(&i>1) %then %let listvars=&listvars,;
    %let listvars=&listvars "%sysfunc(varname(&dsid,&i))";
    %end;
%let buhbye=%sysfunc(close(&dsid));
%let totvars=1;
%let byvars_comma_quote=;
%do %while(%length(%scan(&sortvars,&totvars,,S))>1);
    %if &totvars>1 %then %let byvars_comma_quote=&byvars_comma_quote,
           "%scan(&sortvars,&totvars,,S)";
    %else %let byvars_comma_quote=&byvars_comma_quote
           "%scan(&sortvars,&totvars,,S)";
    %let totvars=%eval(&totvars+1);
    %end;
data _null_;
    if 0 then set &dsn;
    declare hash hashy (dataset:"&dsn (firstobs=&obs1 obs=&obs2)", ordered:'Y');
    hashy.definekey (&byvars_comma_quote);
           hashy.definedata (&listvars);
           hashy.definedone ();
           hashy.output(dataset:"&dsnout");
    stop;
run;
%if %eval(&syscc>0) %then %let sorterr=&syscc;
%mend;

%macro safesort (meth1= /* primary sorting algorithm (SORT, SQL, HASH) */,
    meth2= /* secondary sorting algorithm (SORT, SQL, HASH) */,
    dsn= /* input data set name in DSN or LIB.DSN format */,
    dsnout= /* output data set name in DSN or LIB.DSN format */,
    sortvars = /* space-delimited list of sort variables */,
    thresh = /* file size (in MB) to implement multi-step sort */);
%global safesorterr;
%let safesorterr=;
%global filesize;
%global obstot;
%local i;
%if %length(%scan(&dsn,2,0.))=0 %then %let lib=work;
%else %do;
    %let lib=%scan(&dsn,1,0.);
    %let dsn=%scan(&dsn,2,0.);
    %end;
%if %length(%scan(&dsnout,2,0.))=0 %then %let libout=work;
%else %do;
    %let libout=%scan(&dsnout,1,0.);
    %let dsnout=%scan(&dsnout,2,0.);
    %end;
proc sql noprint;
    select filesize format=15.
```

```
    into :filesize
    from dictionary.tables
    where libname="%upcase(&lib)" and memname="%upcase(&dsn)";
    quit;
run;
proc sql noprint;
    select count(*) format=15.
    into :obstot
    from &lib..&dsn
    quit;
run;
%put FILESIZE: &filesize;
%put OBSTOT: &obstot;
* specify additional business rules here that initiate a multi-step sort;
%if %sysevalf(&filesize/1048576<&thresh) %then %do;
    %put TRADITIONAL METHOD;
    %&meth1(dsn=&lib..&dsn, dsnout=&libout..&dsnout, obs1=1, obs2=&obstot,
            sortvars=&sortvars);
    %end;
%if %eval(&syscc>0) %then %let thresh=%sysevalf(&thresh/2);
%if %sysevalf(&filesize/1048576>=&thresh) or %eval(&syscc>0) %then %do;
    %put OTHER METHOD;
    %let groups=%sysevalf((&filesize/1048576)/&thresh,ceil);
    %do i=1 %to &groups;
           %put GROUPNOW: &i;
           %if &i=1 %then %let obs1=1;
           %else %let obs1=%sysevalf(&obs2+1);
           %if &i=&groups %then %let obs2=&obstot;
           %else %let obs2=%sysevalf(&i*&obstot/&groups,ceil);
           %put FILEOBS: &obs1;
           %put OBS: &obs2;
           %&meth2(dsn=&lib..&dsn, dsnout=&libout..tempsort&i, obs1=&obs1,
                  obs2=&obs2, sortvars=&sortvars);
           %end;
    %if %length(&sorterr)>0 %then %let safesorterr=&sorterr;
    data &libout..&dsnout;
           set
           %do i=1 %to &groups;
                  &libout..tempsort&i
                  %end;;
           by &sortvars;
    run;
    proc datasets library=%upcase(&libout) nolist;
    %do i=1 %to &groups;
           delete tempsort&i;
           %end;
    run;
    %if &syscc>0 %then %let safesorterr=&syscc;
    %end;
quit;
%mend;
```

APPENDIX B. TOWEROFBABEL MACRO

```
* a new sample data set can be created with the NEWTOWER parameter YES;
* but to continue to build on the current data set, set NEWTOWER to NO
options replace; /* required to iteratively construct the data set;

%macro towerofbabel(dsn = /* data set name in LIB.DSN format */,
    newtower = /* YES to delete current DSN and create a new one */,
    numvar = /* number of numeric variables created */,
    charvar = /* number of character variables created */,
    obs = /* number of observations being created */);
%let syscc=0;
* create data set if it does not exist;
%if %sysfunc(exist(&dsn))=0 or &newtower=YES %then %do;
    data &dsn;
           length
           %if %eval(&numvar>0) %then %do i=1 %to &numvar;
                  num&i 8
                  %end;
           %if %eval(&charvar>0) %then %do i=1 %to &charvar;
                  char&i $20
                  %end;;
           format
           %if %eval(&numvar>0) %then %do i=1 %to &numvar;
                  num&i 10.
                  %end;
           %if %eval(&charvar>0) %then %do i=1 %to &charvar;
                  char&i $20.
                  %end;;
           if _n_=1 then delete;
    run;
    %end;
data x (drop=obs j); /* for large data sets, this should not be placed
           in the WORK library */
    if 0 then set &dsn;
    call streaminit(123);
    do obs=1 to &obs;
           %if %eval(&numvar>0) %then %do i=1 %to &numvar;
                  num&i=int(rand('uniform')*1000000000); *from 1 to 1 billion;
                  %end;
           %if %eval(&charvar>0) %then %do i=1 %to &charvar;
                  char&i='';
                  do j=1 to 20;
                         char&i=cats(char&i,byte(int(rand('uniform')*26)+65));
                                *A through Z;
                         end;
                  %end;
           output;
           end;
run;
proc append base=&dsn data=x;
run;
%mend;
```