PharmaSUG 2018 – Paper AD-07

Pinching Off Your SAS® Log: Adapting from Loquacious to Laconic Logs To Facilitate Near-Real Time Log Parsing, Performance Analysis, and Dynamic, Data-Driven Design and Optimization

Troy Martin Hughes

ABSTRACT

Too often in SAS® literature, the role of the program log is narrowly conceptualized as a static, post hoc quality control review that validates program success through the detection of program failure or the lack thereof. Especially when software development occurs outside of a formalized software development life cycle (SDLC), as is often the case with non-production software and within end-user development environments, SAS practitioners must painfully parse logs in search of notes, warnings, runtime errors, and other often elusive indications of functional failure. A substantial body of SAS literature advances antediluvian manual log review through the automation of log parsing and analysis and subsequent communication of program success (or failure) to stakeholders. To a lesser extent, SAS literature depicts how log parsing can be utilized to extract, analyze, and ultimately improve software performance metrics. After a cursory review of SAS automated log parsing literature, this text elucidates and expands this second objective of automated log parsing, demonstrating how performance metrics can be analyzed in near-real time to drive program flow dynamically. By pinching off shorter logs and saving these as temporary text files, SAS programs can analyze performance metrics for individual procedures or processes, enabling software to detect anomalous or undesirable CPU, input/output (I/O), or memory consumption and to respond dynamically to optimize execution.

INTRODUCTION

SAS practitioners are often introduced to the SAS log as the lone method through which SAS programs are validated; if warnings and runtime errors are absent from the log, the program is assumed to have executed correctly. Logic errors and other latent defects could of course exist without the presence of warnings or runtime errors, but these are typically uncovered only through a thorough understanding of the underlying data, resultant data products, and the code that transforms the former into the latter. Thus, the first toddling steps in a SAS career are typically consumed by executing snippets of code and immediately evaluating the log for indications of success or failure. This "code and fix" method is expected from neophytes in any programming language, and even from experienced developers who incrementally develop and empirically test in rapid succession, but is insufficient for stable, production software that demands both higher reliability and integrity.

Thus, as production software is approved and released or as software performance requirements increase, reliance on the SAS log for program validation and failure detection should commensurately decrease in favor of more reliable and efficient methods. For example, production software intended to be automated, scheduled, and run at regular intervals cannot rely on SAS practitioners to babysit it, checking its log and changing its diapers whenever it has an accident. For this reason, dozens of SAS publications are dedicated to demonstrating automated log analysis. Too often in SAS literature, however, automated log analysis is depicted as the primary or only method to detect warnings, runtime errors, and other notes or exceptions. While post hoc log analysis can *detect* exceptions after software completion or termination, it fails to *handle* exceptions because by the time detection has occurred, it is too late for the software to respond dynamically.

For example, rather than implementing a log parsing algorithm that detects the SAS note that is created when division-by-zero occurs, a more useful solution always is to detect the division-by-zero exception programmatically during software execution or to prevent the exception by detecting the state (i.e., zero denominator) that produces it. Thereafter, business rules can prescribe the appropriate action to handle the exception, which may occur behind the scenes without log alert or stakeholder notification, thus reducing gratuitous notes that would otherwise be printed to the log. In contrast, most SAS literature demonstrates automated log parsing that generates an exception report (in the form of a data set, log output, static report, dashboard update, or email) that describes or summarizes issues discovered in one or more log files. For example, a parsing solution might summarize the location and details of all division-by-zero exceptions found in a log file or across several log files. However, the "automated" report thereafter typically requires manual review by SAS practitioners, thus impeding the objective of automation.

A more practical method to drive reliability and incorporate validation into production software is to implement exception handling routines that detect warnings, runtime errors, and other exceptions through return code and error code analysis. This type of exception management is preferred over ad hoc log analysis—even when the latter is automated—because abnormal events, states, and environments can be detected and handled as they occur and according to prescribed business rules. For example, SAS global macro variables such as &SYSERR, &SYSCC, &SQLRC, or &SYSRC—as well as user-defined return codes—can be evaluated programmatically during execution to validate program success or to detect its failure without reliance on the log. In separate texts, the author introduces SAS software exception handling[i] as well as a toolbox of techniques through which software can handle exceptions[ii].

While automated log parsing is primarily depicted in SAS literature as facilitating program validation and failure detection, to a lesser extent, it demonstrates the evaluation of SAS performance metrics to gauge system resource (i.e., CPU, I/O, memory) utilization. A handful of publications demonstrate the analysis of SAS FULLSTIMER metrics to measure, compare, or aggregate the performance of SAS processes. For example, performance analysis is essential in comparing the relative efficiency of competing programmatic methods, in demonstrating performance variance through repeated measures analysis, and in the identification of outliers that can indicate or presage performance failure. Similar to automated log parsing for program validation, however, automated log parsing for performance analysis unfortunately is also narrowly demonstrated in SAS literature as a post hoc analysis tool.

This text aims to right this wrong by introducing automated log parsing to facilitate near-real time software performance analysis. By pinching off SAS logs immediately after a single procedure or DATA step, subsequent analysis can drive program flow based on system resource utilization. For example, if the relative efficiency of software decreases below an established threshold, exception handling could prescribe that the software should terminate prematurely or limit parallel processes. Or, if I/O processing increases beyond an established threshold, exception handling could prescribe that the software limit data throughput, increase parallel processing by spawning additional SAS sessions, or take some other preventative or corrective action. Thus, near-real time performance analysis can enable software to respond adroitly to its environment to facilitate performance tuning and optimization.

AUTOMATED LOG ANALYSIS TO SUPPORT PROGRAM VALIDATION AND FAILURE DETECTION

Scores of white papers over the past two decades have demonstrated the inherent benefits of automating the analysis of SAS log files, namely the increased efficiency and reliability with which program success and performance can be (indirectly) assessed. In all examples, the SAS log is written to a text file—with the PRINTTO procedure, the LOG parameter of the SYSTASK statement, or the LOG parameter when SAS is spawned from the operating system (OS)—after which the text file is parsed by the same or a separate SAS program. The vast majority of log parsing examples detect warnings, runtime errors, and other exceptions in the log to validate program success. Subsequent exception reports may be used to analyze, aggregate, or communicate log results, but stakeholders typically must still review these reports manually.

Log analysis to demonstrate program success or failure can be a critical component of a software quality control plan. A robust, scalable, and flexible SAS macro can interrogate logs strewn across directories to ferret out warnings, runtime errors, troublesome notes, and other indications of program failure. SAS support even provides its own SAS Log Error Checking Tool that produces a synthesized report of warnings and runtime errors detected in SAS logs[iii]. Log analysis is especially essential where SAS code, inputs, environmental attributes, or other factors are likely to fluctuate and, for this reason, is heavily relied upon during software development and testing phases of the SDLC. While automated log parsing should not be the primary method used to validate program success or detect its failure in stable, production software, it can provide a valuable quality control safety net. For example, some organizations and industries mandate that log files not only be parsed but also archived to facilitate auditing.

The following literature review highlights programs that automatically parse log files to validate software success, in all cases by searching for keywords such as "warning" or "error" or other user-defined words and phrases. Only texts that demonstrate log file analysis are included, so solutions such as Derek Morgan's SAS Component Language (SCL) program (which emails SAS logs to stakeholders when their programs fail) are excluded[iv]. Other texts that demonstrate only the addition of user-defined comments to the SAS log, such as Lafler and Rosenbloom's solution, can improve log readability but are similarly excluded from this review[v]. Finally, texts that instead aim to assess SAS performance rather than functional success are described later in this text.

In chronological order, white papers that demonstrate automated analysis of SAS logs to support failure detection and validation of software execution include:

- Lauren Haworth (1997) demonstrates parsing logs to search for keywords such as "error" or "uninitialized" and then prints selected lines using the GPRINT procedure[vi].
- Li and Troxell (2001) demonstrate a DATA step that reads all log files in a directory and saves only those lines that contain "note", "warning", or "error"[vii].
- Carey Smoak (2002) describes the CHECKLOG macro that searches a log file for keywords and phrases such as "warning", "error", or "repeats of BY values" and prints these lines to the log[viii]. Unfortunately, the macro is not included in the text.
- Malachy Foley (2004) demonstrates a program that searches for "info", "warning", "error", and "note" in the log after which selected lines are printed to the log[ix].
- MaryAnne Hope (2004) searches log files for "suspicious messages" in the log, including notes, warnings, and runtime errors, after which a data set is produced containing the exceptions[x].
- Adel Fahmy (2004) demonstrates a phenomenal log parsing tool comprised of several modular macros that search for keywords and phrases located in a separate user-defined text file[xi]. This scalable approach ensures customizable reporting without the need to alter the underlying code. His 2010 text is largely a reprisal of this 2004 work so it definitely should be read but is not included in Table 1[xii].
- Aaron Augustine (2006) demonstrates the LOGCHECK macro that parses a log file for keywords such as "warning" or "error", creates a report, and emails this alert to stakeholders[xiii].
- Kevin Lee (2007) demonstrates a solution that saves log files, searches for keywords "warning" or "error", and prints those lines to the SAS log[xiv].
- Milorad Stojanovic (2008) demonstrates a program that parses a log file in search of notes, warnings, and runtime errors, and displays a summary report[xv].
- Suzanne Humphreys (2008) demonstrates the macro LOGCHECK that searches a specified directory for log files, parses all logs in search of notes, warnings, and runtime errors, and summarizes these in a report[xvi].
- Fang and Gorrell (2009) demonstrate output from a utility program (a single-line batch file) that searches log files for keywords such as "warning" or "error" and prints selected lines in a command prompt window[xvii].
- Amit Baid (2009) demonstrates the macro SCANLOG that scans a user-specified directory then parses all log files in search of a keywords and phrases such as "warning", "error", or "has 0 observations[xviii]".
- Dodlapati, Karidi, and Vanam (2010) discuss various warnings and notes common in SAS logs as well as the benefit of identifying these automatically in log files; however, the discussion is purely theoretical and not accompanied by code so it is omitted from Table 1[xix].
- Salman Ali (2010) demonstrates how Perl regular expressions can be used to parse log files in search of the expression "Transfer Complete" to indicate successful FTP transfer[xx]. If the phrase does not appear in the log, stakeholders are emailed an alert and failed processes are resubmitted automatically. This rare example highlights log parsing that drives dynamic processing rather than simply producing exception reports that must be subsequently examined by stakeholders.

- Ronald Palanca (2011) demonstrates the MPR_CHECKLOG macro that parses a log file for keywords and phrases such as "warning", "error", or "invalid data" and prints an exception report to the interactive SAS window[xxi].

- Vaessen, Pannemans, Quesada, and Vyverman (2011) demonstrate a log parser that searches a log for the keywords "note", "warning", and "error" after which the results are color-coded and posted within a web portal[xxii].

- Matthew Psioda (2012) demonstrates a short program that parses logs for "warning" or error" and prints results to the log[xxiii].

- Christopher Schacherer (2012) demonstrates a log parsing process that searches for "error" within a SAS log and, if discovered, alerts stakeholders via email[xxiv].

- Qiling Shi (2012) demonstrates a program that searches for "note", "warning", or "error" within SAS log files and displays the results with the SUMMARY procedure[xxv].

- Jack Hamilton (2012) demonstrates automatic log parsing through regular expressions that identify keywords and phrases such as "warning", "error", and "stopped processing this step", after which the results are printed to a report[xxvi].

- Yogesh Pande (2012) demonstrates a macro LOG_CHECK that scans a directory for all log files, parses the files for keywords or phrases, and displays results in a color-coded report[xxvii].

- Chris Swenson (2012) showcases a highly customizable CHECKLOG macro that searches log files for a list of keywords, saves the results in an exception data set, and optionally emails a summary report to stakeholders[xxviii].

- Brit Miner (2013) demonstrates his self-proclaimed "fully endowed" solution that parses log files for warnings or runtime errors and which can terminate the program if prerequisite processes fail[xxix]. The solution optionally can automatically email results to stakeholders, thus facilitating dynamic responses to issues that are encountered.

- Emmy Pahmer (2014) demonstrates a solution that parses a log file for warnings, runtime errors, and specific notes, after which the results are saved and printed to a report[xxx].

- Mohan and Vijayarangan (2014) demonstrate parsing log files to detect notes, warnings, and runtime errors that are encountered[xxxi]. They produce sophisticated log analysis and visualization, including Excel pivot tables and graphs that convey exceptions encountered by type and frequency.

- Rasheed and Vijayarangan (2014) demonstrate a SAS program that runs asynchronously (i.e., concurrently) and analyzes log files while they are still being created[xxxii]. The CHASETHELOG macro can send an email to stakeholders as warnings or runtime errors are detected—that is, before program termination—and can even terminate the offending program via the KILL statement, thus demonstrating a rare example of dynamic processing facilitated by near-real time log analysis.

- RIchann "Pretty in Pink" Watson (2017) demonstrates the macro CHECKLOGS that scans a folder for log files that match a parameterized naming convention, after which the log files are parsed for warning and error terms[xxxiii]. This dynamic solution is highly customizable and enables specific log files to be isolated and parsed independently or in cohorts through the parameterized macro invocation.

Table 1 further differentiates the previous texts by highlighting features observed across seven functional characteristics, including:

- **Multi** - Multiple log files can be analyzed through a single invocation of the log parsing macro or program. Most commonly, a file search is performed on a folder (and/or its subfolders) after which the files are iteratively parsed. In other cases, individual log files are concatenated but retain sufficient information to identify their respective SAS programs. One of the principal benefits of multi-log solutions is the ability to press a button, analyze limitless logs, and often create a comprehensive report that identifies all log exceptions appearing within a folder or across a SAS environment.

- **Words** - These solutions flexibly enable users to modify log search terms without modification to the code itself. Single-word terms must be included or modified either through dynamic macro parameters or by modifying external text files that contain the keywords. Thus, solutions that parse log files only through if-then-else or other hardcoded conditional logic statements are not included.
- **Phrases** - These solutions flexibly enable users to modify log search terms by adding single words or phrases without modification to the code itself. This characteristic differs from the previous “Words” characteristic because these solutions can parse phrases (having spaces). Similar to the “Words” characteristic, these solutions also must utilize search terms that are modified either through dynamic macro parameters or external text files. Thus, solutions that parse log files for words (or phrases) through if-then-else or other hardcoded conditional logic statements are not included.
- **Data Set** - A data set is produced that includes warnings, runtime errors, or other issues identified in the log file, usually as the penultimate step before an exception report is generated. In a few examples, however, no data set is created and log file exceptions are printed directly to the log or an interactive SAS window.
- **Report** - These examples produce an exception report, typically with the REPORT or PRINT procedures.
- **Email** - These solutions automatically generate an email alert to one or more stakeholders when certain warning or error conditions are discovered in a log file.
- **Asynch** - Rasheed and Vijayarangan (2014) stand alone in this category as the only authors to provide an inventive approach in which log files are incrementally queried even as they are still being created. As soon as a warning, runtime error, or other offensive event is detected, the log parser—running from a separate SAS session and continuously interrogating the expanding log file—can terminate the program producing the log. This asynchronous automation ensures that subsequent, derivative code can be intelligently skipped rather than becoming embroiled in cascading failures.

Author	Multi	Words	Phrases	Data Set	Report	Email	Asynch
Haworth (1997)				yes	yes		
Li and Troxell (2001)	yes			yes			
Smoak (2002)							
Foley (2004)				yes			
Hope (2004)				yes			
Fahmy (2004)	yes	Yes	yes	yes	yes		
Augustine (2006)				yes	yes	yes	
Lee (2007)				yes			
Stojanovic (2008)				yes	yes		
Humphreys (2008)	yes			yes	yes		
Fang and Gorrell (2009)							
Baid (2009)	yes			yes	yes		
Ali (2010)	yes				yes	yes	
Palanca (2011)				yes	yes		
Vaessen, Pannemans, Quesada, and Vyverman (2011)				yes	yes		
Psioda (2012)	yes						
Schacherer (2012)				yes		yes	
Shi (2012)				yes	yes		
Hamilton (2012)	yes			yes	yes		
Pande (2012)	yes			yes	yes		
Swenson (2012)	yes	Yes		yes	yes	yes	
Minor (2013)	yes					yes	
Pahmer (2014)				yes	yes		
Mohan and Vijayarangan (2014)	yes			yes	yes		
Rasheed and Vijayarangan (2014)				yes	yes	yes	yes
Watson (2017)	yes			yes	yes		

Table 1. Differentiating Functionality among SAS Automated Log Parsing Literature

AUTOMATED LOG ANALYSIS TO SUPPORT PERFORMANCE ANALYSIS

As demonstrated in the previous section, the majority of automated log parsing solutions alert users to warnings, runtime errors, and other exceptions detected in SAS logs. A less common yet arguably more appropriate use of automated log parsing analyzes software performance metrics saved to log files. The FULLSTIMER SAS system option elicits maximum verbosity of system resource utilization although the specific FULLSTIMER metrics generated will vary by OS[xxxiv]. For example, page faults or page reclaims might be displayed in FULLSTIMER metrics within a UNIX—but not a Windows—environment. SAS documentation introduces the FULLSTIMER option and demonstrates how its metrics can be used to evaluate and improve software performance[xxxv]. SAS documentation also demonstrates the graphical analysis of FULLSTIMER metrics with nmon and perfmon freeware tools[xxxvi].

Whereas log parsing solutions that validate program success (and detect its failure) are numerous and represent an array of methods and functionality, far less diversity exists among log parsing solutions that assess program performance. Performance metrics are few and finite and in all examples are saved to a SAS data set or to macro variables for subsequent analysis. Thus, the log parsing solution demonstrated herein (PINCHLOG) does not differ substantively from past literature in the way that performance metrics are generated or retrieved but only in the context in which these metrics are analyzed to drive dynamic program flow. The following literature review includes only examples in which performance metrics are extracted from log files. For example, texts that demonstrate timing SAS processes by subtracting program start time from completion time are not included, whereas examples that time processes through analysis of log file real time or CPU time metrics are included.

In chronological order, white papers that demonstrate automated analysis of SAS logs to support system resource utilization and performance evaluation include:

- Robert Patten (2003) demonstrates the ALGTEST macro that compares performance metrics of competing versions of code, for example, to demonstrate which of several strategies is most efficient[xxxvii]. The solution incrementally captures real time and CPU time metrics and saves these to a data set for further analysis.

- Michael Raithel (2005) authored the most publicized and comprehensive performance analysis tool, the LOGPARSE macro[xxxviii]. Prominently featured on the SAS Support website, this portable solution parses FULLSTIMER metrics and saves them to a data set for further analysis. Ronald Fehd (2006) later makes subtle improvements to the LOGPARSE macro in his text[xxxix].

- Middela and Bhamidipati (2008) introduce the COMPARE macro that analyzes performance metrics and is intended to summarize and aggregate metrics between two competing versions of SAS code[xl].

- LeRoy Bessler (2010) demonstrates software that measures system resource utilization and emails stakeholders if CPU usage crosses an established threshold[xli]. This added context is important, as it demonstrates how dynamic processing can be driven by performance metric evaluation.

- Steven First (2012) demonstrates use of the SCAPROC procedure to analyze performance metrics saved to a log file[xlii]. SCAPROC is not further discussed in this text but is described in the Base SAS 9.4 Procedures Guide[xliii].

- Lingqun Liu (2016) demonstrates automated log analysis using a two-step parsing process that extracts real time and CPU time from log files[xliv].

PINCHLOG INVOCATION AND EXCEPTION HANDLING

PINCHLOG is unextraordinary in its ability to extract and save performance metrics; in fact, it is more limited than some of the referenced solutions because it is intended to be run only after a single procedure or DATA step. The contextual implementation of PINCHLOG is unique, however, in its aim to facilitate dynamic performance evaluation during software execution. PINCHLOG parses FULLSTIMER performance metrics immediately after a DATA step or procedure has completed, after which the program can appropriately respond. PINCHLOG is derived from the READFULLSTIMER macro found in the author's text but also optionally creates a metrics data set for later analysis[xlv].

To run the examples in this text, the following &PATH assignment must be modified to reflect the folder where the programs within this text will be saved:

```
%let path=D:\sas\pinchlog; * location must be modified;
```

Thereafter, the programs in Appendices A and B (pinchlog.sas and makedata.sas) should be saved in the &PATH folder. The LIB library is defined and the PINCHLOG and MAKEDATA macros are initialized with the following code:

```
libname lib "&path";
%include "&path\pinchlog.sas";
%include "&path\makedata.sas";
```

The MAKEDATA macro creates a sample, randomized data set that is ideal for performance testing, load testing, stress testing, or repeated measures testing. The sample data set can contain a parameterized number of character and/or numeric fields (incremented as char1, char2, num1, num2, etc.) that vary in length based on parameterized input. Because not only the number of observations but also the number of *unique* observations is a critical predictive performance indicator in many procedures such as SORT or FREQ, not only the number of observations but also the number of *unique* values can be specified. Within this text, MAKEDATA is repeatedly used to efficiently create simulated data for analysis. The MAKEDATA macro definition follows:

```
%macro makedata(dsn= /* data set name in LIB.DSN or DSN format */,
   obs= /* number of observations */,
   obsuni= /* number of unique observations */,
   charvar=1 /* number of character variables */,
   charlen=10 /* length of character variables */,
   numvar=0 /* number of numeric variables */,
   numlen=0 /* length of numeric variables (3 to 8) */);
```

The PINCHLOG macro definition follows:

```
%macro pinchlog(logfile= /* file path, name, and extension of log analyzed */,
   dsnmetrics= /* optional metrics data set in LIB.DSN or DSN format */,
   othervars= /* optional tokenized list of user-defined variables */);
```

PINCHLOG includes the following parameters:

- LOGFILE (Required) – the file path, file name, and file extension of the log file being parsed.
- DSNMETRICS (Optional) – the data set name of the metrics data set that is optionally modified or, if it does not already exist, automatically created. If no data set is specified, the OTHERVARS parameter is ignored.
- OTHERVARS – this parameter includes a tokenized list of custom variables that can be additionally saved to the metrics data set. For example, specifying the following partial code will create two user-defined variables Obs and Procedure and their respective LENGTH, FORMAT, and LABEL statements. Note that dynamic values can be injected (as in the case of &OBS) by including macro variables:

  ```
  othervars=(var=obs, val=&obs, len=8, form=8., lab=Observation Count /
     var=procedure, val=incremental sort, len=$20, form=$20., lab=Procedure)
  ```

  The VAR, VAL, and LEN sub-parameters are required when the OTHERVARS parameter is specified while the FORM and LAB sub-parameters are optional. The entire OTHERVARS parameter must be enclosed within parentheses, with commas separating all sub-parameters and a slash (/) separating each user-defined variable that is specified. The FORM sub-parameter requires a period just as in the FORMAT statement. Quotation marks are not needed in the LAB or VAL sub-parameters, even if the VAL sub-parameter represents a character variable. Parentheses, slashes, and commas cannot be used in the LAB sub-parameter and, because the PINCHLOG macro does not include exception handling to validate the OTHERVARS parameter, the specified format must be followed precisely to avoid unhandled exceptions.

The following sample code directs the log to the &LOGFILE text file, executes the SORT procedure, and subsequently invokes PINCHLOG to parse the log file. Note that the metrics data set is not created in this example:

```
options fullstimer;

* create bogus data set;
data somedata;
```

```
   length char1 $5;
   char='x';
run;

%let logfile=&path\log.txt;
proc printto log="&logfile" new;
run;
proc freq data=somedata;
   tables char1;
run;
proc printto; * optional;
run;

%pinchlog(logfile=&logfile);

%put REALTIME / MEMORY: &realtime / &memory;
```

Invoking PINCHLOG parses the log file, extracts performance metrics, and initializes these into numerous global macro variables described in the following section. The second PRINTTO procedure is optional and does not affect PINCHLOG; however, the log should be redirected somewhere or else it will continue to grow unnecessarily in the &LOGFILE log file until it is overwritten. If a subsequent PINCHLOG invocation followed the first, a corresponding PRINTTO procedure would also be required but could again specify &LOGFILE, thus overwriting and recycling the same log file repetitively. In this initial example, the FULLSTIMER Realtime and Memory statistics are printed to the log while in later examples FULLSTIMER metrics are be analyzed to drive program flow.

One disadvantage of recycling log files is that they cannot be archived for auditing purposes; a failure could occur, be written to a temporary log file, be overwritten by a subsequent PINCHLOG invocation, and effectively vanish within seconds without evidence of the failure. To ensure that exceptions do not go undetected, PINCHLOG includes basic log validation that detects warnings and runtime errors and records these in the PINCHLOGRC (PINCHLOG return code) global macro variable. This exception handling is demonstrated in the following example, which iteratively sorts an incrementally larger data set but which terminates if the SORT procedure fails or exceeds two seconds:

```
%macro incremental_sort(dsn= /* data set name in LIB.DSN format */,
   dsnout= /* sorted data set name in LIB.DSN format */,
   dsnmetrics= /* data set name for metrics table */,
   minobs= /* minimum number of observations */,
   maxobs= /* maximum number of observations */,
   addobs= /* number of observations to add each iteration */,
   charlen= /* length of character field created */,
   logfile= /* temporary log file analyzed by PINCHLOG */);
%local i;
%do i=&minobs %to &maxobs %by &addobs;
   %put NOW SORTING: &i;
   %makedata(dsn=&dsn, obs=&i, obsuni=&i, charvar=1, charlen=&charlen);
   proc printto log="&logfile" new;
   run;
   %let syscc=0;
   proc sort data=&dsn out=&dsnout;
      by char1;
   run;
   proc printto;
   run;
   %pinchlog(logfile=&logfile, dsnmetrics=&dsnmetrics,
      othervars=(var=obs, val=&i, len=8, form=8., lab=Observation Count /
      var=charlen, val=&charlen, len=8, form=8., lab=Character Length));
   %if &syscc^=0 %then %do;
      %put INCREMENTAL_SORT Failure;
      %return;
      %end;
   %if &pinchlogRC^=0 or &pinchlogchildRC^=0 %then %do;
      %put PINCHLOG Failure;
      %return;
      %end;
   %if %sysevalf(&realtime>2) %then %do; * arbitrary threshold;
      %put Time Limit Exceeded;
      %return;
      %end;
   %end;
%mend;

%incremental_sort(dsn=test, dsnout=testsorted, dsnmetrics=metrics,
   minobs=1000000,  maxobs=100000000, addobs=1000000, charlen=10,
   logfile=&path\sortlog.txt);
```

In this example, the INCREMENTAL_SORT macro iteratively creates and sorts 100 data sets that range from one million to 100 million observations, and simulates the type of load and stress testing that might be required before deploying the SORT procedure in production ETL software. The global macro variable &REALTIME is created by PINCHLOG, represents the elapsed time during SORT execution, and is used to ensure that the program is stopped if a single SORT exceeds two seconds—i.e., defined arbitrarily as a performance failure. Functional failures are also detected via &SYSCC. Because &SYSCC is initialized to 0 immediately before the SORT procedure, this ensures that any warning or runtime error found during subsequent evaluation of &SYSCC could have occurred only during the current iteration that is executing. Upon examination of the Metrics data set, the OBS field will include the number of observations created during that iteration. Additionally, if the macro terminates prematurely because the SORT time threshold is exceeded, the Realtime metric in the Metrics data set will demonstrate this arbitrary exception.

The &PINCHLOGRC macro variable detects warnings or runtime errors that can occur during execution of PINCHLOG, for example, if a log file is specified that does not exist. The &PINCHLOGCHILDRC is not utilized here but detects warnings or runtime errors *detected* by PINCHLOG rather than *caused* by PINCHLOG. Thus, if the SORT procedure had failed, this exception would not only be directly detectable through &SYSCC but also indirectly detectable through &PINCHLOGCHILDRC because “error” would be discovered in the log file. The &PINCHLOGCHILDRC return code is useful when PINCHLOG is reading a log file that the parent process did not produce, in which case neither &SYSCC nor &SYSERR would be able to detect the failure directly. Both &PINCHLOGRC and &PINCHLOGHILDRC are set to 0 when no warning or runtime error is detected.

In load and stress testing, rather than terminating a loop, macro, or program when a warning or runtime error is detected, the exception could instead be captured in the metrics data set as a user-defined variable specified with the OTHERVARS parameter. This method is demonstrated in the final example in this text and ensures that the functional failure is salient during subsequent analysis of the metrics data set. This type of analysis is a critical component of stress testing in which the primary objective is to push software until functional or performance failure.

PINCHLOG METRICS AND MACRO VARIABLES

Table 2 demonstrates the FULLSTIMER metrics analyzed by PINCHLOG and includes the OS in which the metric can be observed, the SAS global macro variable into which the metric is initialized (also the variable optionally created in the metrics data set), and the unit of measurement.

Log Text	**OS**	**Macro Variable / Data Set Variable**	**Units**
real time	Win, UNIX	realtime	seconds

user cpu time	Win, UNIX	usercputime	seconds
system cpu time	Win, UNIX	systemcputime	seconds
cpu time	Win, UNIX	cputime	seconds
memory	Win, UNIX	memory	MB
os memory	Win, UNIX	osmemory	MB
step count	Win, UNIX	stepcount	steps
switch count	WIN, UNIX	switchcount	switches
page faults	UNIX	pagefaults	faults
page reclaims	UNIX	pagereclaims	reclaims
voluntary context switches	UNIX	volcontextswitches	switches
involuntary context switches	UNIX	involcontextswitches	switches
block input operations	UNIX	blockinops	operations
block output operations	UNIX	blockoutops	operations

Table 2. FULLSTIMER Performance Metrics as Parsed by PINCHLOG

To reiterate, PINCHLOG differs from most automated log solutions because its intent is to capture performance metrics rather than identify functional failure. Secondarily, it differs because the performance metrics are intended to be analyzed immediately after completion or termination of a procedure or DATA step. The following example illustrates stress testing that might be used to capture parallel processing performance metrics. By incrementing the number of parallel processes, the optimal range of concurrent processes can be established—i.e., the upper and lower limits for the optimal number of processes to run concurrently. The following program should be saved as Child_sort.sas in the &PATH folder:

```
%let path=D:\sas\pinchlog;
libname lib "&path";
options fullstimer;

proc sort data=lib.somedata out=lib.somedatasorted&sysparm;
   by char1;
run;
```

Note that because the child program will be run from a separate SAS session, it must stand alone and include all library definitions, macro variable initializations, system options, %INCLUDE statements, and other information that is session-specific. This highlights the reliability, integrity, and efficiency gained when static and/or frequently used system information is defined automatically within AUTOEXEC.sas and configuration files. The following program, which iteratively spawns Child_sort.sas, should be saved as Stresstesting.sas in the &PATH folder:

```
libname lib "&path";
%include "&path\pinchlog.sas";
%include "&path\makedata.sas";

%macro stresstesting(dsn=lib.somedata,
   dsnout=lib.somedatasorted,
   dsnmetrics= /* metrics data set name in LIB.DSN or DSN format */,
   minprocesses= /* minimum number of processes */,
   maxprocesses= /* maximum number of processes */);
%local i j ratio childerr maxcputime maxrealtime;
```

```
%do i=&minprocesses %to &maxprocesses;
   %let childerr=0;
   %makedata(dsn=lib.somedata, obs=10000000, obsuni=10000,
      charvar=1, charlen=10);
   %do j=1 %to &i;
      systask command """%sysget(SASROOT)\sas.exe"" -noterminal -nosplash
         -sysin ""&path\child_sort.sas""
         -log ""&path\child_sort&j..txt"" -sysparm ""&j"""
         taskname=task&j status=rc&j;
      %end;
   waitfor _all_
      %do j=1 %to &i;
         task&j
         %end;;
   * check for SYSTASK failure in child processes (1=warning, 2=error);
   %do j=1 %to &i;
      %if &&&rc&j=1 or &&&rc&j=2 %then %let childerr=&&&rc&j;
      %end;
   %let maxcputime=0;
   %let maxusertime=0;
   %do j=1 %to &i;
      %pinchlog(logfile=&path\child_sort&j..txt, dsnmetrics=&dsnmetrics,
         othervars=(var=processes, val=&i, len=8, form=8., lab=Processes /
         var=childerr, val=&childerr, len=8, form=8,. lab=Child Error Code));
      %if %sysevalf(&usercputime>&maxcputime)
         %then %let maxcputime=&usercputime;
      %if %sysevalf(&realtime>&maxrealtime)
         %then %let maxrealtime=&realtime;
      %end;
   %put PROCESSES: &i  MAXREAL: &maxrealtime  REAL/PROCESSES:
      %sysevalf(&maxrealtime/&i);
   %if %sysevalf((&maxcputime/&maxrealtime)<.5) %then %do;
      %put Too many processes--SORT has been terminated;
      %return;
      %end;
   %end;
%put ALL DONE!;
%mend;

%stresstesting(dsnmetrics=lib.metrics_sort, minprocesses=1, maxprocesses=24);
```

The STRESSTESTING macro spawns from between one and 24 parallel processes with the asynchronous SYSTASK statement. It aims to determine when the incremental addition of parallel processes begins to reduce performance. In this example, relative efficiency is operationalized as the ratio of User CPU Time to Realtime (&USERCPUTIME to &REALTIME), given that this ratio will decrease as more concurrent processes are running on the system and consuming resources. Thus, on the first iteration, only one SAS session (and batch process) is spawned so only one set of metrics is generated and recorded in the metrics data set. On the second iteration, however, two SAS sessions are spawned, two sets of metrics are generated and saved, but only the highest User CPU Time and Realtime are initialized as &MAXCPUTIME and &MAXREALTIME, respectively, to calculate the ratio.

When the SAS system is running single-threaded processes efficiently, the ratio of User CPU time to Realtime will be very close to but slightly less than 1.0. However, because multithreaded processes (like SORT) do more at once, the ratio of User CPU Time to Realtime should exceed 1.0. This ratio can be compared under different conditions (e.g., altering the number of parallel processes running) or over time (e.g., when repeated measures analysis is performed). A SAS system taxed with concurrent processes—including SAS programs as well as unrelated processes such as the infamous virus detection software—will produce lower performance ratios. Table 3 demonstrates abridged metrics generated in the LIB.Metrics_sort data set.

Real Time	User CPU Time	System CPU Time	Memory in MB	OS Memory in MB	Processes
2.53	4.07	0.18	623.24	624.66	1
2.98	4.87	0.34	623.24	624.66	2
3.01	4.82	0.20	623.24	624.66	2
3.75	5.20	0.28	623.24	624.66	3
3.65	5.28	0.35	623.24	624.66	3
3.67	5.40	0.20	623.24	624.66	3

Table 3. Sample Metrics from Stress Test of Parallel SORT Procedures: First Three Iterations

In this example, the performance ratio of the first iteration is 1.61 (or 4.07 seconds / 2.53 seconds), the second 1.62 (or 4.87 seconds / 3.01 seconds), and the third 1.44 (or 5.4 seconds / 3.75 seconds). However, by the 15th iteration (when 15 SORT procedures are simultaneously running), the ratio had dropped to 0.49 (or 8.25 seconds / 16.89 seconds) so the loop was halted (per the arbitrary 0.5 threshold) with a user-defined error message. In other words, if software needed to sort 15 data sets in series, it would take approximately 37.95 seconds (or 2.53 seconds x 15); however, by sorting 15 data sets concurrently, all data sets sort in only 16.89 seconds at a tremendous time savings.

However, if the %RETURN statement is removed so that STRESSTESTING completes 24 iterations despite crossing the 0.5 ratio threshold, the relative inefficiency of processing by the final iteration is clearly visible. By the 24th iteration, the ratio of User CPU Time to Realtime had dropped to 0.33 (or 8.43 seconds / 25.91 seconds). Thus, 24 data sets can be sorted in series in approximately 60.72 seconds (or 2.53 seconds x 24) while the same data sets can be sorted in parallel in 25.91 seconds. This is not to say that each of the 24 data sets took 25 seconds, only that the slowest process to complete took 25 seconds. Thus, a dependent process waiting on all SORT procedure to complete would need to wait until the slowest SORT had completed.

To determine when the optimal number of processes is reached (which maximizes SORT performance by decreasing runtime) the ratio of maximum Realtime to the number of concurrent processes can be evaluated over time. Figure 1 demonstrates that by the time five or six processes are running concurrently, the system has reached maximum throughput at which point additional processes do not improve performance. Additional concurrent processes may, however, deplete system resources (e.g., memory) so one proven strategy is to utilize performance testing to ensure that the number of parallel processes does not exceed the point at which performance is optimized.

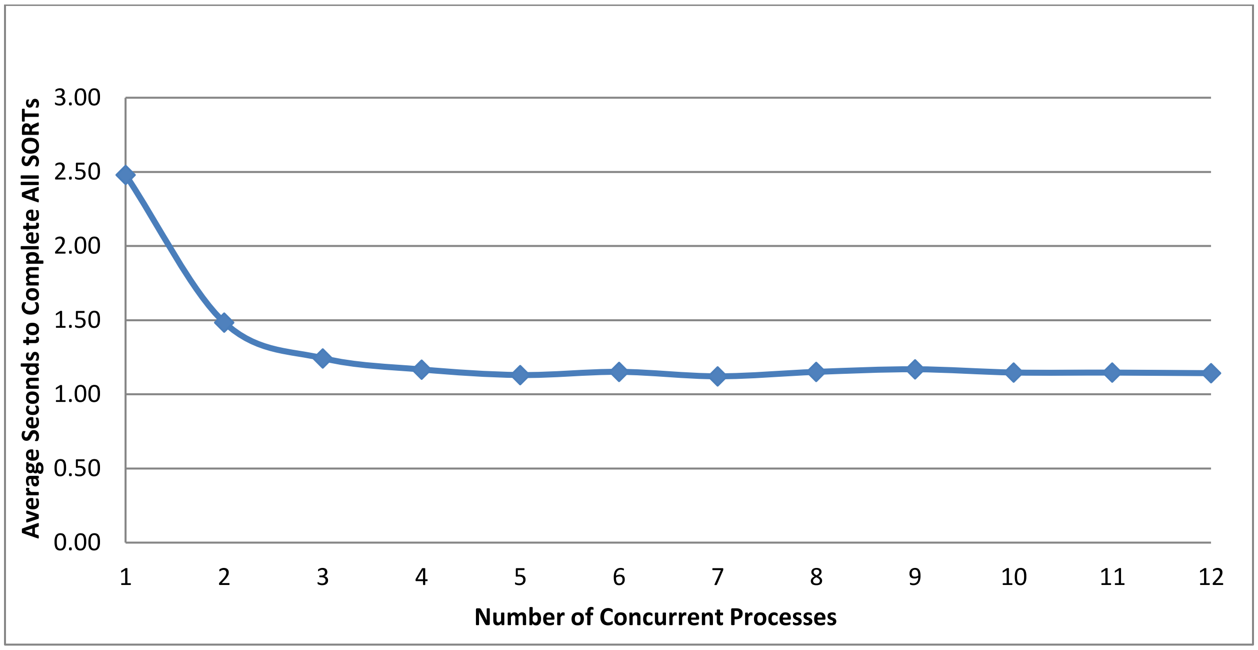

Figure 1. Average Time To Complete SORT Procedure While Incrementing Parallel Processes

While parallel processing is often a sure-fire way to improve performance, it must be done responsibly to ensure that the optimal number of processes is selected. Too many processes running concurrently will cause performance degradation or software to grind to a halt with CPU or memory failure. Stress testing should be an essential quality assurance step during software testing to ensure the optimal number of processes is selected. Hardware and infrastructure variability across SAS environments dictate that load testing and stress testing must be performed in an environment that most closely facsimiles that in which the software will be deployed for production. PINCHLOG can be implemented to optimize software as it runs (such as by dynamically modifying the number of parallel processes invoked) and to monitor system resource utilization to prevent functional or performance failure.

USING PINCHLOG FOR REPEATED MEASURES PERFORMANCE ANALYSIS

Although demonstrating ad hoc rather than dynamic analysis, PINCHLOG can be implemented to facilitate repeated measures performance analysis. Because runtime and resource utilization can fluctuate wildly based on environmental conditions (e.g., memory availability or the number of concurrent processes running on a system), assessing performance metrics multiple times can produce more accurate results that depict not only “typical” performance but also performance outliers and measures of performance variability.

For example, to gain a more accurate representation of the performance of the SORT procedure in a specific SAS environment, the following code iterates through ten successive sorts and utilizes PINCHLOG to capture FULLSTIMER metrics:

```
%macro repeatedmeasures(dsn= /* data set name in LIB.DSN or DSN format */,
   dsnout= /* output data set name in LIB.DSN or DSN format */,
   logfile= /* path, file name, and extension of log to be analyzed */,
   dsnmetrics= /* metrics data set in LIB.DSN or DSN format */,
   iterations= /* number of iterations to run */,
   obs= /* number of observations */,
   obsuni= /* approximate number of unique observations */,
   charlen= /* length of character variable being FREQqed */);
%local i;
%do i=1 %to &iterations;
   %let syscc=0;
```

```
      %put ITERATION: &i;
      %makedata(dsn=&dsn, obs=&obs, obsuni=&obsuni, charvar=1, charlen=&charlen);
      proc printto log="&logfile" new;
      run;
      proc freq data=&dsn noprint;
         tables char1 / out=&dsnout;
      run;
      proc printto;
      run;
      %pinchlog(logfile=&logfile, dsnmetrics=&dsnmetrics);
      %end;
   %put ALL DONE!;
   %mend;

   %repeatedmeasures(dsn=lib.freqdata, dsnout=lib.freqout,
      dsnmetrics=lib.freqmetrics, logfile=&path\freqlog.txt,
      iterations=10, obs=1000000, obsuni=1000, charlen=16);
```

Easy-peasy and a performance metrics data set is created automatically that can be used for post hoc analysis, for example, to determine a more accurate mean and median for FREQ Realtime. However, a more dynamic example might additionally include an analysis of the Metrics data set after each iteration to assess the variability of performance metrics and halt the loop when some threshold was achieved, for example, if it was determined that a representative data sample had been generated.

USING PINCHLOG FOR COMPARATIVE PROCESS PERFORMANCE ANALYSIS

A final use case for PINCHLOG involves comparing system performance of competing processes, for example, comparing sort performance of the SORT procedure and SQL procedure. Repeated measures analysis has also been implemented to provide some assurance that accurate metrics are produced. The following macro iteratively runs the SORT procedure and SQL procedure to sort a sample data set:

```
   %macro sortvssql (dsn= /* data set name in LIB.DSN or DSN format */,
      dsnout= /* output data set name in LIB.DSN or DSN format */,
      logfile= /* path, file name, and extension of log to be analyzed */,
      dsnmetrics= /* metrics data set in LIB.DSN or DSN format */,
      iterations= /* number of iterations to run */,
      obs= /* number of observations */,
      obsuni= /* approximate number of unique observations */,
      charlen= /* length of character variable being FREQqed */);
   %local i filesize cpucount memsize;
   * convert to GB;
   %let memsize=%sysevalf(%sysfunc(getoption(xmrlmem))/(1024*1024*1024));
   %let cpucount=%sysfunc(getoption(cpucount));
   * LIB required so PROC SQL can obtain filesize;
   %if %scan(&dsn,1,.)=0 %then %let dsn=WORK.&dsn;
   %if %scan(&dsnout,1,.)=0 %then %let dsnout=WORK.&dsnout;
   %do i=1 %to &iterations;
      %makedata(dsn=&dsn, obs=&obs, obsuni=&obsuni, charvar=1, charlen=&charlen);
      proc sql noprint;
         select filesize format=15.
         into :filesize
         from dictionary.tables
         where libname="%scan(%upcase(&dsn),1,.)" and
            memname="%scan(%upcase(&dsn),2,.)";
      quit;
      %let filesize=%sysevalf(&filesize/(1024*1024)); * convert to MB;
   * PROC SORT;
      %let syscc=0;
      proc printto log="&logfile" new;
      run;
      proc sort data=&dsn out=&dsnout;
         by char1;
      run;
      proc printto;
      run;
      %pinchlog(logfile=&logfile, dsnmetrics=&dsnmetrics,
         othervars=(var=procedure, val=PROC SORT, len=$20, form=$20.,
                 label=Test Procedure /
            var=err, val=&syscc, len=8, form=8., lab=SYSCC error code /
            var=obs, val=&obs, len=8, form=comma15., lab=Obs /
            var=obsuni, val=&obsuni, len=8, form=comma15., lab=Unique Obs /
            var=charlen, val=&charlen, len=8, form=8., lab=Character Length /
            var=filesize, val=&filesize, len=8, form=comma15.2, lab=Size in MB /
            var=cpucount, val=&cpucount, len=8, form=8., lab=CPUCOUNT /
            var=memsize, val=&memsize, len=8, form=8.2, lab=MEMSIZE in GB));
      %if &pinchlogRC^=0 or &pinchlogchildRC^=0 %then %do;
         %put Something smells funny...;
         %return;
         %end;
   * PROC SQL sort;
      %let syscc=0;
      proc printto log="&logfile" new;
      run;
      proc sql noprint;
         create table &dsnout AS
            select * from &dsn
            order by char1;
         quit;
      proc printto;
      run;
      %pinchlog(logfile=&logfile, dsnmetrics=&dsnmetrics,
         othervars=(var=procedure, val=PROC SQL SORT, len=$20, form=$20.,
                 label=Test Procedure /
            var=err, val=&syscc, len=8, form=8., lab=SYSCC error code /
            var=obs, val=&obs, len=8, form=comma15., lab=Obs /
            var=obsuni, val=&obsuni, len=8, form=comma15., lab=Unique Obs /
            var=charlen, val=&charlen, len=8, form=8., lab=Character Length /
            var=filesize, val=&filesize, len=8, form=comma15.2, lab=Size in MB /
            var=cpucount, val=&cpucount, len=8, form=8., lab=CPUCOUNT /
            var=memsize, val=&memsize, len=8, form=8.2, lab=MEMSIZE in GB));
      %if &pinchlogRC^=0 or &pinchlogchildRC^=0 %then %do;
         %put Something smells funny...;
         %return;
         %end;
      %end;
   %put ALL DONE!;
   %mend;
   %sortvssql(dsn=lib.somedata, dsnout=lib.somedatasorted,
      dsnmetrics=lib.sortmetrics, logfile=&path\logtemp.txt,
      iterations=20, obs=10000000, obsuni=100000, charlen=16);
```

The macro invocation produces a data set with 40 observations, alternating between PROC SORT and PROC SQL SORT metrics. Additional system parameters CPUCOUNT and XMRLMEM have been captured with the GETOPTIONS function and initialized into respectively named variables within the metrics data set. This allows the software execution conditions to be understood (and replicated) more readily. Moreover, file size is saved into the Filesize variable for use in regression testing (not shown) that might demonstrate the impact of file size on CPU Time, Realtime, or other performance metrics.

Table 1 demonstrates sample output from the UNIVARIATE procedure and the clear performance advantage of using the SQL procedure over the SORT procedure *for this specific example only*—that is, with this data set, on this system, and under these conditions. Thus, while load and stress testing methods and programs can and should often be shared among environments to promote more consistent and comparable metrics, the results will differ from one system and circumstance to the next.

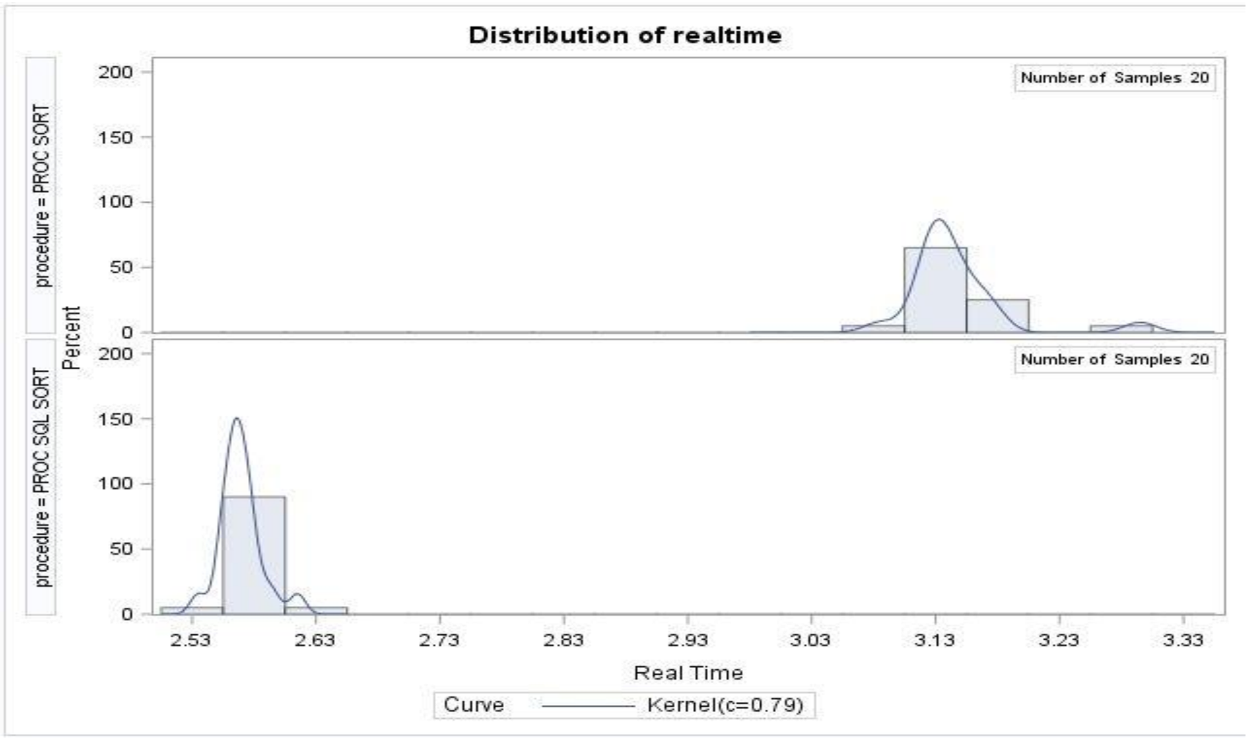

Figure 2. Relative Performance (Realtime) of SORT Procedure to SQL Procedure

To more broadly compare the performance of SORT and SQL, the impact of other characteristics such as observation count must be analyzed. The flexibility of PINCHLOG (and the SQLVSSORT macro in which it was implemented in this example) ensures the extensible solution can be utilized for subsequent performance testing. For example, with only a few more lines of code, this final macro 1) compares performance of the SORT and SQL procedures, 2) assesses performance variability of each method through repeated measures analysis, and 3) increments the number of observations sorted from ten million to 100 million to show the influence of observation count:

```
%macro test();
%local cnt;
%do cnt=10000000 %to 100000000 %by 10000000;
   %sortvssql(dsn=lib.somedata, dsnout=lib.somedatasorted,
      dsnmetrics=lib.sortmetrics_incremental, logfile=&path\logtemp.txt,
      iterations=20, obs=&cnt, obsuni=100000, charlen=16);
   %end;
%mend;
```

```
        %test;
```

While the results are not demonstrated, this example conceptually illustrates how SAS practitioners can utilize PINCHLOG to facilitate univariate and multivariate influence on system and process performance. In a separate text, the author illustrates multiple factors that influence SORT procedure performance, including number of observations, number of unique observations, pre-sort order of observations, file completeness, file compression, number of BY variables, type of BY variables, length of BY variables, file size, available memory, SORTSIZE system option, available CPUs, and SAS interface (e.g., SAS Display Manager vs. SAS University Edition) on which the SORT is performed[xlvi].

Thus, PINCHLOG can provide the heavy lifting to capture and aggregate performance metrics for later analysis while near-real time analysis of PINCHLOG performance metrics can simultaneously ensure that stress testing does not enter an unhealthy state by consuming too many system resources. In a separate text, the author demonstrates ad hoc use of PINCHLOG to collect FULLSTIMER performance metrics to facilitate comparison between and regression testing of the FREQ procedure and a faster alternative, the FREQFAST macro[xlvii].

CONCLUSION

This text introduced the PINCHLOG macro that extracts performance metrics from SAS log files for both immediate analysis and aggregation within a performance metrics data set. By pinching off a SAS log after process completion, subsequent processes can respond dynamically to improve performance and to detect and prevent unhealthy system resource utilization. Moreover, PINCHLOG can be implemented with ease to facilitate post hoc performance analysis such as comparative process testing, load testing, stress testing, and repeated measures analysis.

REFERENCES

[i] Troy Martin Hughes. Why Aren't Exception Handling Routines Routine? Toward Reliably Robust Code through Increased Quality Standards in Base SAS. *Midwest SAS Users Group (MWSUG) 2014*. Retrieved from https://www.mwsug.org/proceedings/2014/BB/MWSUG-2014-BB17.pdf.

[ii] Troy Martin Hughes. Ushering SAS® Emergency Medicine into the 21st Century: Toward Exception Handling Objectives, Actions, Outcomes, and Comms. *Midwest SAS Users Group (MWSUG) 2015*. Retrieved from https://www.mwsug.org/proceedings/2015/PH/MWSUG-2015-PH-08.pdf.

[iii] SAS Log Error Checking Tool. *SAS Support*. SAS Institute, Inc. Cary, NC. Retrieved from http://support.sas.com/kb/44/852.html.

[iv] Derek Morgan. DISTRESS and PATCH: SCL to Support Remote Applications. *SAS Users Group International (SUGI) 2002*. Retrieved from http://www2.sas.com/proceedings/sugi27/p035-27.pdf.

[v] Mary F. O. Rosenbloom and Kirk Paul Lafler. Best Practices: Put More Errors and Warnings in My Log, Please! *SAS Global Forum 2013*. Retrieved from http://support.sas.com/resources/papers/proceedings13/350-2013.pdf.

[vi] Lauren Haworth. Reports Based on SAS Output: Taking Advantage of PROC PRINTTO, Data Steps and PROC GPRINT. *SAS Users Group International (SUGI) 1997*. Retrieved from http://www2.sas.com/proceedings/sugi22/ADVTUTOR/PAPER43.PDF.

[vii] Tianshu Li and John K. Troxell. A Macro To Report Problematic SAS Log Messages in a Production Environment. *Northeast SAS Users Group (NESUG) 2001*. Retrieved from http://www.lexjansen.com/nesug/nesug01/cc/cc4008.pdf.

[viii] Carey G. Smoak. A Utility Program for Checking SAS Log Files. *SAS Users Group International (SUGI) 2002*. Retrieved from http://www2.sas.com/proceedings/sugi27/p096-27.pdf.

[ix] Malachy J. Foley. Cutting the SAS® LOG Down to Size. *Southeast SAS Users Group (SESUG) 2004*. Retrieved from http://analytics.ncsu.edu/sesug/2004/SY05-Foley.pdf.

[x] MaryAnne D. Hope. The Automatic Detection of Problems in the SAS Log. *Western Users of SAS Software (WUSS) 2004*. Retrieved from http://www.lexjansen.com/wuss/2004/coders_corner/c_cc_the_automatic_detection.pdf.

[xi] Adel Fahmy. Program Validation: Logging the Log. *Northeast SAS Users Group (2004)*. Retrieved from http://www.lexjansen.com/nesug/nesug04/ap/ap09.pdf.

[xii] Adel Fahmy. Logging the Log Magic: Pulling the Rabbit out of the Hat. *PharmaSUG 2010*. Retrieved from http://www.lexjansen.com/pharmasug/2010/TT/TT08.pdf.

[xiii] Aaron Augustine. You've Got E-Mail: Automatic Log Checking Via E-Mail Notification. SAS Users Group International (SUGI) 2006. Retrieved from http://www2.sas.com/proceedings/sugi31/128-31.pdf.

[xiv] Kevin Lee. How to QC Your Own Programs. *Northeast SAS Users Group (NESUG) 2007*. Retrieved from http://www.lexjansen.com/nesug/nesug07/ap/ap12.pdf.

[xv] Milorad Stojanovic. SAS Log Summarizer - Finding What's Most Important in the SAS Log. *Southeast SAS Users Group (SESUG) 2008*. Retrieved from http://analytics.ncsu.edu/sesug/2008/CC-037.pdf.

[xvi] Suzanne Humphreys. %LOGCHECK: A Convenient Tool for Checking Multiple Log Files. *PharmaSUG 2008*. Retrieved from http://www.lexjansen.com/pharmasug/2008/cc/CC02.pdf.

[xvii] Zhengyi Fang and Paul Gorrell. A Utility Program for Quickly Identifying Log Error or Warning Messages. *Northeast SAS Users Group (NESUG) 2009*. Retrieved from http://www.lexjansen.com/nesug/nesug09/cc/CC15.pdf.

[xviii] Amit Baid. Catch the Bad Guys!!! A Utility Program To Check SAS Log Files. *PharmaSUG 2009*. Retrieved from http://www.lexjansen.com/pharmasug/2009/po/PO25.pdf.

[xix] Sridhar R Dodlapati, Kiran Kumar Karidi, and Mahipal R Vanam. Log Checking: What To Check and Why? *Northeast SAS Users Group (NESUG) 2010*. Retrieved from http://www.lexjansen.com/nesug/nesug10/cc/cc08.pdf.

[xx] Salman Ali. Pragmatic Approach To Resubmit Failed Jobs in Production Environment. *SAS Global Forum 2010*. Retrieved from http://support.sas.com/resources/papers/proceedings10/006-2010.pdf.

[xxi] Ronald R. Palanca. You've Got ERROR. *SAS Global Forum 2011*. Retrieved from http://support.sas.com/resources/papers/proceedings11/068-2011.pdf.

[xxii] Ralf Vaessen, Danny Pannemans, Juan Quesada Koen Vyverman. In Control of Your Data Warehouse Processes with the Help of SAS Stored Processes and the SAS Information Delivery Portal. *SAS Global Forum 2011*. Retrieved from http://support.sas.com/resources/papers/proceedings11/375-2011.pdf.

[xxiii] Matthew Psioda. Using Windows Batch Files to Sequentially Execute Sets of SAS Programs Efficiently. *Southeast SAS Users Group (SESUG) 2012*. Retrieved from http://analytics.ncsu.edu/sesug/2012/PO-08.pdf.

[xxiv] Christopher W. Schacherer. SAS Data Management Techniques: Cleaning and Transforming Data for Delivery of Analytic Datasets. *Midwest SAS Users Group (MWSUG) 2012*. Retrieved from https://www.mwsug.org/proceedings/2012/DM/MWSUG-2012-DM06.pdf.

[xxv] Qiling Shi. Check and Summarize SASLog Files. *Midwest SAS Users Group (MWSUG) 2012*. Retrieved from https://www.mwsug.org/proceedings/2012/S1/MWSUG-2012-S101.pdf.

[xxvi] Jack Hamilton. What Do You Mean, Not Everyone Is Like Me: Writing Programs for Others To Run. *SAS Global Forum 2012*. Retrieved from http://support.sas.com/resources/papers/proceedings12/229-2012.pdf.

[xxvii] Yogesh Pande. Log Checks Made Easy. *SAS Global Forum 2012*. Retrieved from http://support.sas.com/resources/papers/proceedings12/042-2012.pdf.

[xxviii] Chris Swenson. An Advanced, Multi-Featured Macro Program for Reviewing Logs. *SAS Global Forum 2012*. Retrieved from http://support.sas.com/resources/papers/proceedings12/098-2012.pdf.

[xxix] Brit Miner. Using SAS Driver Programs To Automate Workflows and Respond to the Unexpected. *PharmaSUG 2013*. Retrieved from https://www.pharmasug.org/proceedings/2013/CC/PharmaSUG-2013-CC34.pdf.

[xxx] Emmy Pahmer. Making the Log a Forethought Rather Than an Afterthought. *SAS Global Forum 2014*. Retrieved from http://support.sas.com/resources/papers/proceedings14/1556-2014.pdf.

[xxxi] Palanisamy Mohan and Amarnath Vijayarangan. Automated Log Analyzer Dashboard. *PharmaSUG China 2014*. Retrieved from http://www.lexjansen.com/pharmasug-cn/2014/CC/PharmaSUG-China-2014-CC08.pdf.

[xxxii] Harun Rasheed and Amarnath Vijayarangan. Chasing the Log File While Running the SAS Program. *SAS Global Forum 2014*. Retrieved from http://support.sas.com/resources/papers/proceedings14/1762-2014.pdf.

[xxxiii] Richann Watson. Check Please: An Automated Approach to Log Checking. *SAS Global Forum 2017*. Retrieved from http://support.sas.com/resources/papers/proceedings17/1173-2017.pdf.

[xxxiv] FULLSTIMER SAS Option. *Scalability and Performance*. SAS Institute. Retrieved from http://support.sas.com/rnd/scalability/tools/fullstim/.

[xxxv] Michael Williams. Troubleshoot Your Performance Issues: SAS Technical Support Shows You How. *SAS Global Forum 2009*. Retrieved from http://support.sas.com/resources/papers/proceedings09/333-2009.pdf.

[xxxvi] Tony Brown. SAS Performance Monitoring - A Deeper Discussion. SAS Institute, Inc. *SAS Global Forum 2008*. Retrieved from http://www2.sas.com/proceedings/forum2008/387-2008.pdf.

[xxxvii] Robert Patten. Run Time Comparison Macro. *SAS Users Group International (SUGI) 2003*. Retrieved from http://www2.sas.com/proceedings/sugi28/113-28.pdf.

[xxxviii] Michael A. Raithel. Programmatically Measure SAS Application Performance on Any Computer Platform with the New LOGPARSE SAS Macro. *SAS Users Group International (SUGI) 2005*. Retrieved from http://www2.sas.com/proceedings/sugi30/219-30.pdf.

[xxxix] Ronald J. Fehd. Modifying The LogParse PassInfo Macro To Provide a Link between Product Usage in Rtrace Log and Time Used in Job Log. *Southeast SAS Users Group (SESUG) 2006*. Retrieved from http://analytics.ncsu.edu/sesug/2006/AP14_06.PDF.

[xl] Sreekanth Reddy Middela and Venkata Sekhar Bhamidipati. Benchmark Macro %COMPARE. *Northeast SAS Users Group (NESUG) 2008*. Retrieved from http://www.lexjansen.com/nesug/nesug08/cc/cc17.pdf.

[xli] LeRoy Bessler. More Ways to Use SAS to Manage, Monitor, and Control SAS or the SAS BI Server: Tools for the SAS User, Server Administrator, or Manager. *SAS Global Forum 2010*. Retrieved from http://support.sas.com/resources/papers/proceedings10/279-2010.pdf.

[xlii] Steven First. The SAS Log: A Wealth of Data and Job Flow Information. *SAS Global Forum 2012*. Retrieved from http://support.sas.com/resources/papers/proceedings12/237-2012.pdf.

[xliii] SCAPROC Procedure. Base SAS 9.4 Procedures Guide, Seventh Edition. SAS Institute, Inc. Cary, NC. Retrieved from http://support.sas.com/documentation/cdl/en/proc/70377/HTML/default/viewer.htm#n05aazp6jtoup0n1qjee3h7jto24.htm.

[xliv] Lingqun Liu. SAS Advanced Programming with Efficiency in Mind: A Real Case Study. *Midwest SAS Users Group (MWSUG) 2016*. Retrieved from https://www.mwsug.org/proceedings/2016/BB/MWSUG-2016-BB18.pdf.

[xlv] Troy Martin Hughes. *SAS Data Analytic Development: Dimensions of Software Quality*. John Wiley and Sons, Inc. Hoboken, NJ. 2016.

[xlvi] Troy Martin Hughes. Sorting a Bajillion Records: Conquering Scalability in a Big Data World. *Southeast SAS Users Group (SESUG) 2015*. Retrieved from http://support.sas.com/resources/papers/proceedings16/11888-2016.pdf.

[xlvi] Troy Martin Hughes. From FREQing Slow to FREQing Fast: Facilitating a Four-Times-Faster FREQ with Divide-and-Conquer Parallel Processing. *Southeast SAS Users Group (SESUG) 2017*.

CONTACT INFORMATION

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

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

APPENDIX A. PINCHLOG MACRO

```
%macro pinchlog(logfile= /* path, file name, and extension */,
   dsnmetrics= /* optional metrics data set in LIB.DSN or DSN format */,
   othervars= /* optional tokenized list of user-defined variables */);
* all times converted from HH:MM:SS.ss to SSSS.xx format;
%let syscc=0;
%global pinchlogRC pinchlogchildRC realtime usercputime systemcputime
   memory osmemory stepcount switchcount pagefaults pagereclaims
   volcontextswitches involcontextswitches blockinops blockoutops;
%let pinchlogRC=99;
%let pinchlogchildRC=0;
%let realtime=.;
%let usercputime=.;
%let systemcputime=.;
%let memory=.;
%let osmemory=.;
%let stepcount=.;
%let switchcount=.;
%let pagefaults=.;
%let pagereclaims=.;
%let volcontextswitches=.;
%let involcontextswitches=.;
%let blockinops=.;
%let blockoutops=.;
data _null_;
   length tab $500;
   infile "&logfile" truncover;
   input tab $500.;
   if strip(tab)=:'WARNING' or strip(tab)=:'ERROR' then do;
      call symput('pinchlogchildRC','4');
      return;
      end;
   if _n_>=7 then do; * skips the metrics produced by PRINTTO itself;
      if lowcase(substr(tab,1,9))='real time' then do;
         if count(scan(substr(tab,10),1,' '),':')=0 then
            call symput('realtime',scan(substr(tab,10),1,' '));
         else if count(scan(substr(tab,10),1,' '),':')=1 then
            call symput('realtime',put(((input(strip(scan(substr(tab,10),
            1,':')),8.0) * 60) +
            input(strip(scan(substr(tab,10),2,':')),8.2)),8.2));
         else if count(scan(substr(tab,10),1,' '),':')=2 then
            call symput('realtime',put(((input(strip(scan(substr(tab,10),
            1,':')),8.0) * 3600) +
            (input(strip(scan(substr(tab,10),2,':')),8.0) * 60) +
            input(strip(scan(substr(tab,10),3,':')),8.2)),8.2));
         end;
      else if lowcase(substr(tab,1,13))='user cpu time' then do;
         if count(scan(substr(tab,14),1,' '),':')=0 then
            call symput('usercputime',scan(substr(tab,14),1,' '));
         else if count(scan(substr(tab,14),1,' '),':')=1 then
            call symput('usercputime',put(((input(strip(scan(substr(tab,14),
            1,':')),8.0) * 60) +
            input(strip(scan(substr(tab,14),2,':')),8.2)),8.2));
         else if count(scan(substr(tab,14),1,' '),':')=2 then
            call symput('usercputime',put(((input(strip(scan(substr(tab,14),
            1,':')),8.0) * 3600) +
            (input(strip(scan(substr(tab,14),2,':')),8.0) * 60) +
            input(strip(scan(substr(tab,14),3,':')),8.2)),8.2));
         end;
      else if lowcase(substr(tab,1,15))='system cpu time' then do;
         if count(scan(substr(tab,16),1,' '),':')=0 then
            call symput('systemcputime',scan(substr(tab,16),1,' '));
         else if count(scan(substr(tab,16),1,' '),':')=1 then
            call symput('systemcputime',put(((input(strip(scan(substr(tab,16),
            1,':')),8.0) * 60) +
            input(strip(scan(substr(tab,16),2,':')),8.2)),8.2));
         else if count(scan(substr(tab,16),1,' '),':')=2 then
            call symput('systemcputime',put(((input(strip(scan(substr(tab,16),
            1,':')),8.0) * 3600) +
            (input(strip(scan(substr(tab,16),2,':')),8.0) * 60) +
            input(strip(scan(substr(tab,16),3,':')),8.2)),8.2));
         end;
      * convert KB to MB;
      else if lowcase(substr(tab,1,6))='memory' then
         call symput('memory',put(input(scan(substr(tab,7),1,'
            k'),8.3)/1024,8.3));
      else if lowcase(substr(tab,1,9))='os memory' then
         call symput('osmemory',put(input(scan(substr(tab,10),1,'
            k'),8.3)/1024,8.3));
      else if lowcase(substr(tab,1,10))='step count' then do;
         call symput('stepcount',scan(substr(tab,11),1,' '));
         call symput('switchcount',scan(substr(tab,11),4,' '));
         end;
      else if lowcase(substr(tab,1,11))='page faults' then
         call symput('pagefaults',scan(substr(tab,12),1,' '));
      else if lowcase(substr(tab,1,13))='page reclaims' then
         call symput('pagereclaims',scan(substr(tab,14),1,' '));
      else if lowcase(substr(tab,1,26))='voluntary context switches' then
         call symput('volcontextswitches',scan(substr(tab,27), 1,' '));
      else if lowcase(substr(tab,1,28))='involuntary context switches' then
         call symput('involcontextswitches',scan(substr(tab,29),1,' '));
      else if lowcase(substr(tab,1,24))='block input operations' then
         call symput('blockinops',scan(substr(tab,25),1,' '));
      else if lowcase(substr(tab,1,25))='block output operations' then
         call symput('blockoutops',scan(substr(tab,26),1,' '));
      end;
run;
* optionally initialize user-defined variables;
* at least VAR, VAL, and FORM sub-parameters are required for each variable;
%if %length(othervars)>0 %then %do;
   %local otherval otherlen otherform otherlab var val len form lab i j;
   %let otherval=;
   %let otherlen=;
   %let otherform=;
   %let otherlab=;
   %let othervars=%sysfunc(compress(%bquote(&othervars),
      %nrstr(%))%nrstr(%(),));
   %let i=1;
   %do %while(%length(%scan(%bquote(&othervars),&i,/))>1);
      %let var=;
      %let val=;
      %let len=;
      %let form=;
      %let lab=;
      %let varstring=%scan(%bquote(&othervars),&i,/);
      %let j=1;
      %do %while(%length(%scan(%bquote(&varstring),&j,%str(,)))>1);
         %let valstring=%scan(%bquote(&varstring),&j,%str(,));
         %if %lowcase(%scan(&valstring,1,=))=var
            %then %let var=%lowcase(%scan(&valstring,2,=));
         %if %lowcase(%scan(&valstring,1,=))=val
            %then %let val=%lowcase(%scan(&valstring,2,=));
         %if %lowcase(%scan(&valstring,1,=))=len
            %then %let len=%lowcase(%scan(&valstring,2,=));
         %if %lowcase(%scan(&valstring,1,=))=form
            %then %let form=%lowcase(%scan(&valstring,2,=));
         %if %lowcase(%scan(&valstring,1,=))=lab
            %then %let lab=%scan(&valstring,2,=);
         %let j=%eval(&j+1);
         %end;
      %if %length(&var)>0 and %length(&len)>0 and %length(&val)>0 %then %do;
         %let otherlen=&otherlen &var &len;
         %if %substr(&len,1,1)=$ %then
            %let otherval=&otherval &var="&val"%str(;);
         %else %let otherval=&otherval &var=&val%str(;);
         %if %length(&form)>0 %then %let otherform=&otherform &var &form;
         %if %length(&lab)>0 %then %let otherlab=&otherlab &var="&lab";
         %end;
      %let i=%eval(&i+1);
      %end;
   %end;
* optionally create/modify metrics data set;
%if %length(&dsnmetrics)>0 %then %do;
   %if %sysfunc(exist(&dsnmetrics))=0 %then %do;
      data &dsnmetrics;
         length realtime 8 usercputime 8 systemcputime 8 memory 8 osmemory 8
            stepcount 8 switchcount 8 pagefaults 8 pagereclaims 8
            volcontextswitches 8 involcontextswitches 8 blockinops 8;
         format realtime 8.2 usercputime 8.2 systemcputime 8.2 memory 8.3
            osmemory 8.3 stepcount 8. switchcount 8. pagefaults 8.
            pagereclaims 8. volcontextswitches 8. involcontextswitches 8.
            blockinops 8.;
         label realtime='Real Time' usercputime='User CPU Time'
            systemcputime='System CPU Time' memory='Memory in MB'
            osmemory='OS Memory in MB' stepcount='Step Count'
            switchcount='Switch Count' pagefaults='Page Faults'
            pagereclaims='Page Reclaims' volcontextswitches=
            'Voluntary Context Switches' involcontextswitches=
            'Involuntary Context Switches' blockinops='Block Input Operations'
            blockoutops='Block Output Operations';
         %if %length(&otherval)>0 %then %do;
            length &otherlen;;
            %if %length(&otherform)>0 %then %do;
                  format &otherform;;
                  %end;
            %if %length(&otherlab)>0 %then %do;
                  label &otherlab;;
                  %end;
            %end;
         if not missing(realtime);
      run;
      %end;
   data pinchtemp;
      if 0 then set &dsnmetrics;
      realtime=&realtime;
      usercputime=&usercputime;
      systemcputime=&systemcputime;
      memory=&memory;
      osmemory=&osmemory;
      stepcount=&stepcount;
      switchcount=&switchcount;
      pagefaults=&pagefaults;
      pagereclaims=&pagereclaims;
      volcontextswitches=&volcontextswitches;
      involcontextswitches=&involcontextswitches;
      blockinops=&blockinops;
      %if %length(&otherval)>0 %then %do;
         &otherval;
         %end;
      output;
   run;
   proc append base=&dsnmetrics data=pinchtemp;
   run;
   %end;
%let pinchlogRC=&syscc;
%mend;
```

APPENDIX B. MAKEDATA MACRO

```
* creates a customizeable, random data set;
%macro makedata(dsn= /* data set name in LIB.DSN format */,
   obs= /* number of observations */,
   obsuni= /* number of unique observations */,
   charvar=1 /* number of character variables */,
   charlen=10 /* length of character variables */,
   numvar=0 /* number of numeric variables */,
   numlen=0 /* length of numeric variables (3 to 8) */);
%let syscc=0;
%global makedataRC;
%let makedataRC=99;
%local i j maxnum;
* maxnum represents the highest number that can be saved;
%if %length(&numvar)>0 %then %do;
   %let maxnum=%sysevalf(32*(256**(&numlen-2)));
   %end;
%else %let numvar=0;
data &dsn (drop=obs obs2 i);
   length rando 8 i 8 obs 8 obs2 8
   %if %eval(&charvar>0) %then %do i=1 %to &charvar;
      char&i $&charlen
      %end;
   %if %eval(&numvar>0) %then %do i=1 %to &numvar;
      num&i &numlen
      %end;
   %str(;);
   %let j=%eval(&obsuni-%sysfunc(mod(&obs,&obsuni)));
   do obs=1 to &obsuni;
      %if %eval(&charvar>0) %then %do i=1 %to &charvar;
         char&i='';
         do i=1 to &charlen;
            char&i=cats(char&i,byte(int(rand('uniform')*10)+65)); *A to J;
            end;
         %end;
      %if %eval(&numvar>0) %then %do i=1 %to &numvar;
         num&i=int(rand('uniform')*&maxnum);
         %end;
      do obs2=1 to ifn(obs<=&j,%sysfunc(floor(%sysevalf(&obs/&obsuni))),
            %sysevalf(%sysfunc(floor(%sysevalf(&obs/&obsuni)))+1));
         rando=rand('uniform');
         output;
         end;
      end;
run;
* ensures data are not only random but also in random order;
proc sort data=&dsn out=&dsn (drop=rando);
   by rando;
run;
%let makedataRC=&syscc;
%mend;
```