Paper 080-2022

Calling for Backup When Your One-Alarm Becomes a Two-Alarm Fire: Developing SAS® Data-Driven Concurrent Processing Models through Control Tables and Dynamic Fuzzy Logic

Troy Martin Hughes

ABSTRACT

In the fire and rescue service, a box alarm (or, simply, "alarm") describes the severity of a fire. As an alarm is elevated from a one-alarm fire to a multi-alarm fire, additional, predetermined resources (e.g., personnel and apparatuses) are summoned to combat the blaze more aggressively. Thus, a five-alarm fire—or its equivalent "five-alarm" Cincinnati chili—represents something extremely hot and dangerous. After extinguishment, and as smoke and embers recede and overhaul begins, fire and rescue resources are released "back into service" so they can be utilized elsewhere if necessary. Essential to managing complex fireground operations, this load balancing paradigm is also common in grid and cloud computing environments in which additional computational resources can be shifted temporarily to an application or process to maximize its performance and throughput. This text instead demonstrates a programmatic approach in which SAS® extract-transform-load (ETL) operations are decomposed and modularized and subsequently directed (for execution) by control tables. If increased throughput is required, additional instances of the ETL program can be invoked concurrently, with each software instance performing various operations on different data sets, thus decreasing overall runtime. A shared control table provides the communications backbone for all SAS sessions by tracking incomplete, in-progress, and completed operations for all data sets. A configuration file allows end users to specify prerequisite processes (that must be completed before an operation can commence), thus facilitating the dynamic fuzzy logic that autonomously selects the specific ETL operations to be executed. This data-driven design approach ensures that the execution of operations can be prioritized, optimized, and, to the extent possible, run in parallel to maximize performance and throughput.

INTRODUCTION

A question often asked of the fire service is "What is a five-alarm fire?" or "What do the respective alarms represent?" When a fire exceeds or is expected to strain the current resources of fire and rescue response, the alarm is elevated, summoning additional, predetermined personnel and apparatuses. Local standard operating procedures (SOPs) specify responses for each alarm, ensuring responsible, adequate, and effective operations. Central to safety and risk mitigation is the notion of "getting ahead of the fire" by anticipating whether additional resources might be needed and by requesting them in advance in a proactive posture. Some incidents such as high-rise fires might require an initial two- or multi-alarm response (as dictated through many SOPs), whereas other incidents might receive a second alarm only after an initial one-alarm response (and subsequent assessment that additional resources are warranted). Although no two fires are identical, the alarm system helps organize a dangerous and potentially chaotic environment while ensuring effective communication and the appropriate allocation of resources.

Less perilous than fireground operations, delays in data processing can instill a similar (albeit diminished) sense of urgency or panic in developers, analysts, and other stakeholders waiting for software to complete. Although firefighters can raise an alarm level to summon additional resources to adapt to changing (and especially worsening) fireground conditions, many SAS enterprises rely on outmoded, serialized extract-transform-load (ETL) processes that operate at a single speed regardless of urgency level. For example, the priority of a transformed data set might skyrocket if a CEO requests a derivative data product by noon, but SAS practitioners are often helpless to expedite the ETL operations that ingest and transform data to produce the required data product. This inability to match increased urgency with increased software performance is frustrating to all stakeholders, with one helpless customer once remarking, "Can't you just run two copies of the ETL software at the same time to process the data twice as fast?!"

And, in some circumstances, a savvy SAS practitioner can do just that—run two or more instances of ETL software concurrently to double, triple, quadruple, or further improve software performance and runtime. By spawning multiple sessions of the SAS Display Manager (aka, the Windowing environment application), and by running the same ETL software concurrently in each session, overall productivity and throughput is increased. More firefighters. More hoses. More water. Less fire. This concurrent processing methodology represents a programmatic approach that requires the decomposition, modularization, and prioritization of ETL processes, as well as a control table to coordinate (and communicate among) SAS sessions. Software complexity is increased, although SAS practitioners who invest in this modest fixed cost are rewarded with tremendously higher-performing software.

SETUP FOR EXAMPLES

The following code initializes the &LOC global macro variable, which should be changed to an appropriate folder on the user's system (in which all subsequent programs and files should be saved):

```
* initialize path and library;
%let loc=D:\sas\etl\; /* USER MUST CHANGE LOCATION */
libname etl "&loc";
```

Note that the &LOC initialization and the LIBNAME statement will need to be copied to subsequent programs, as directed within this text. This duplication is required because programs running separate instances of SAS must separately initialize global macro variables and user-defined libraries. In a real-world example, this redundancy could be avoided by placing the &LOC and ETL initializations within file autoexec.sas file, or by otherwise persisting them on a SAS server.

The LOCKITDOWN macro, demonstrated in Appendix A, should be saved as Lockitdown.sas.

The following code should be run to initialize the three sample data sets referenced throughout this text, having 159, 318, and 477 observations, respectively:

```
* initialize data sets;
data etl.smallfish;
   set sashelp.fish;
run;

data etl.medfish;
   set sashelp.fish sashelp.fish;
run;

data etl.bigfish;
   set sashelp.fish sashelp.fish sashelp.fish;
run;
```

The following code should be run to define the SLEEP macro, which is used to delay the SAS system briefly to simulate elapsed time that would be encountered during actual data processing:

```
* simulates elapsed time during data processing;
%macro sleep(sec);
   %local zzz;
   %let zzz=%sysfunc(sleep(&sec,1));
%mend;
```

FROM MONOLITHIC TO MODULAR DESIGN

As a procedural, fourth-generation language (4GL), SAS is often conceptualized, written, and read like a book—from cover to cover, or start to finish. This sequential flow is natural within data analytic development, in which the objective is typically to retrieve, clean, and transform some data to produce a raw or finished data product. Moreover, ETL process flows typically run without user input or with limited parameterized input at the outset of a batch job, a design that differs substantially from applications that require user input. For this reason, SAS practitioners can tend to write software "books"—monolithic, multi-hundred or even multi-thousand line programs that execute sequentially. In some cases, their length, complexity, and serialization cannot be avoided; however, modularization can often benefit these back-breaking behemoths while increasing software flexibility, reusability, and readability. Moreover, software modularity is the foundation of programmatic concurrent processing models because it enables different modules to be run simultaneously.

The following code depicts monolithic design in which all ETL operations are contained within a single program (ETL.sas); for reference purposes, three distinct processes (i.e., Extract, Transform, and Load) are identified with comments:

```
* program saved as ETL_monolithic.sas;

* extract;
data smallfish_1;
   set etl.smallfish;
run;
%sleep(30);

* transform;
data smallfish_2;
   set smallfish_1;
   length avg_len 8;
   format avg_len 8.3;
   label avg_len='Mean Length';
   avg_len=mean(of length:);
run;
%sleep(20);

* load;
ods html path="&loc" file="smallfish_rpt.html";
title1 "SMALLFISH Report";
proc report data=smallfish_2 (obs=5) nocenter
      style(header)=[color=#FFFFFF backgroundcolor=#696969];
   column species weight avg_len;
run;
ods html close;
%sleep(10);
```

To simulate elapsed time that would occur during actual ETL data processing, the user-defined SLEEP macro is called after each operation. In this first example, Extract completes in 30 seconds, Transform in 20 seconds, and Load in 10 seconds; however, these times could be altered in subsequent examples to change processing outcomes and pathways, as well as the degree of concurrency that can be achieved.

Despite the program representing three distinct operations, the program lacks modularity because these operations are not independently defined. Modularity can be increased by separating these operations into distinct macros and increased further by saving those macros in separate programs. The separation of operations into distinct macros facilitates the ability of software to change the order in which operations are run (given that prerequisites for each operation are still completed first). The subsequent separation of operations into distinct programs moreover increases software stability and integrity because individual modules can be updated without the necessity to access or alter SAS program files in which other operations (i.e., macros) are maintained. To aid in readability, the refactored example (ETL_modular) decomposes the program into separate macros only—not separate program files.

The ETL_monolithic program processes only the Smallfish data set because the data set name is hardcoded within the Extract operation. Because the objective aims to process not only Smallfish but also the Medfish and Bigfish data sets, a more reusable approach should specify the data set name dynamically. To facilitate this software reusability and flexibility, the ETL_modular program parameterizes the data set name in all modules to dynamically reference the temporary data sets and the HTML report that is produced. File names must be unique to ensure that the multiple instances of ETL_modular (that will be running concurrently) do not overwrite data sets or output produced by other instances.

Another hurdle of attaining concurrency across multiple SAS instances is the inability to access the WORK library of other SAS instances. For example, the ETL_monolithic program saved the Smallfish_1 and Smallfish_2 data sets in the WORK library. This is effective in a serialized process flow, but when operations are distributed across multiple SAS instances, a persistent (i.e., named) library to which all SAS instances have access must be utilized. This ensures that all instances of the SAS program running concurrently will be able to access all data sets. To facilitate this objective, the ETL_modular program saves all data sets in the ETL library rather than in WORK.

The following program refactors ETL_monolithic into ETL_modular:

```
* program saved as ETL_modular.sas;

* simulates elapsed time during data processing;
%macro sleep(sec);
   %local zzz;
   %let zzz=%sysfunc(sleep(&sec,1));
%mend;

* extract;
%macro extract(lib= /* library */,
   dsn= /* data set name */);
%let syscc=0;
%global processRC;
%let processRC=GENERAL FAILURE;
data &lib..&dsn._1;
   set &lib..&dsn;
run;
%sleep(30);
%if &syscc=0 %then %let processRC=;
%mend;

* transform;
%macro transform(lib= /* library */,
   dsn= /* data set name */);
%let syscc=0;
%global processRC;
%let processRC=GENERAL FAILURE;
data &lib..&dsn._2;
   set &lib..&dsn._1;
   length avg_len 8;
   format avg_len 8.3;
   label avg_len='Mean Length';
   avg_len=mean(of length:);
run;
%sleep(20);
%if &syscc=0 %then %let processRC=;
%mend;

* load;
* relies on LOC global macro variable;
%macro load(lib= /* library */,
   dsn= /* data set name */);
%let syscc=0;
%global processRC;
%let processRC=GENERAL FAILURE;
ods html path="&loc" file="&dsn._rpt.html";
title1 "%upcase(&dsn) Report";
proc report data=&lib..&dsn._2 (obs=5) nocenter
      style(header)=[color=#FFFFFF backgroundcolor=#696969];
   column species weight avg_len;
run;
ods html close;
%sleep(10);
%if &syscc=0 %then %let processRC=;
%mend;
```

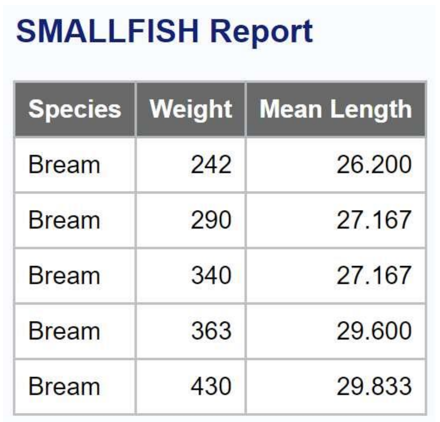
Note that the SLEEP macro must be defined within the program file so that the subsequent EXTRACT, TRANSFORM, and LOAD macros can call it. Also note that basic exception handling has been added by initializing the automatic macro variable &SYSCC to 0 at the start of each macro and by evaluating the value of &SYSCC before the macro terminates. If &SYSCC evaluates to a value greater than zero, this indicates that a warning or runtime error has occurred so the global macro variable &PROCESSRC is returned as “GENERAL FAILURE”; otherwise, &PROCESSRC is set to a Null value. Although not demonstrated, the parent process calling these macros could evaluate the &PROCESSRC return code to determine if a failure had occurred and to respond accordingly.

The EXTRACT, TRANSFORM, and LOAD macros are defined in a single program, yet are not invoked (i.e., called or executed) within this program. A second program—an *engine, driver,* or *controller*—is required to execute these macros, thus allowing greater flexibility because the driver can dynamically determine which macro to run. Because the driver program could be utilized to call other, unrelated ETL programs, this modularity also facilitates the reusability of the driver program.

The following driver program (Driver_small) invokes the EXTRACT, TRANSFORM, and LOAD macros to process the Smallfish data set:

```
* program saved as Driver_small.sas;
%let loc=D:\sas\etl\; /* USER MUST CHANGE LOCATION */
libname etl "&loc";

%include "&loc.etl_modular.sas";

%extract(lib=etl, dsn=smallfish);

%transform(lib=etl, dsn=smallfish);

%load(lib=etl, dsn=smallfish);
```

Table 1 demonstrates the report produced by the LOAD macro and saved as Smallfish_rpt.html.



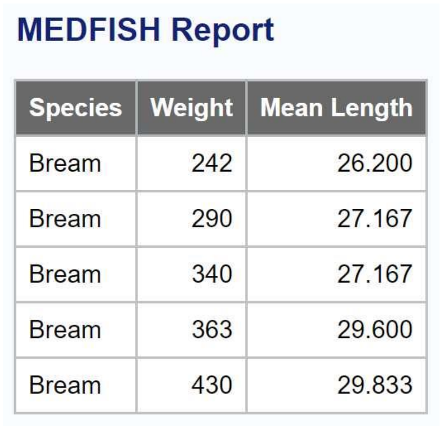
SMALLFISH Report

Species	Weight	Mean Length
Bream	242	26.200
Bream	290	27.167
Bream	340	27.167
Bream	363	29.600
Bream	430	29.833

Table 1. HTML Report Produced by LOAD Macro on Smallfish Data Set

A second driver program (Driver_med) can be similarly created to process the Medfish data set:

```
* program saved as Driver_med.sas;
%let loc=D:\sas\etl\; /* USER MUST CHANGE LOCATION */
libname etl "&loc";

%include "&loc.etl_modular.sas";

%extract(lib=etl, dsn=medfish);

%transform(lib=etl, dsn=medfish);

%load(lib=etl, dsn=medfish);
```


MEDFISH Report

Species	Weight	Mean Length
Bream	242	26.200
Bream	290	27.167
Bream	340	27.167
Bream	363	29.600
Bream	430	29.833

Table 2. HTML Report Produced by LOAD Macro on Medfish Data Set

Note that software reusability is demonstrated in that both data sets are ingested with the same EXTRACT macro, transformed with the same TRANSFORM macro, and are printed with the same LOAD macro. This dynamic functionality is achieved through macro parameters that are supplied from the driver programs. A third driver program (not demonstrated) could be created to process the Bigfish data set. Concurrency has been achieved because both Driver_med and Driver_small can be run simultaneously (in separate SAS sessions). However, a more ideal and maintainable solution would conceptualize a single driver program that processes all data sets, as demonstrated in subsequent sections.

CRITICAL PATH ANALYSIS AND CRITICAL PATH CONFIGURATION FILE

Functional decomposition (or *decomposition*) refers to the practice of identifying discrete functionality within software—that is, identifying modules that (ideally) do one and only one thing—and separating those modules so they can be run and maintained as independently as possible. The previous section demonstrated decomposition in which the ETL_monolithic program was cleaved into three separate macros—EXTRACT, TRANSFORM, and LOAD. This decomposition also separated the program in which the macros reside (ETL_monolithic) from the programs calling them (Driver_small, Driver_medium) to facilitate software reusability and stability.

Decomposition also enables *critical path analysis*—the identification of operational resources and prerequisites. The *critical path* of a program flow describes the operations that must be completed, the order in which those operations must be completed (by accounting for prerequisite operations or activities that must have occurred), and the resources required (e.g., shared access to data sets, exclusive access to data sets, CPUs). Critical path analysis is essential in structuring and ordering program flow and can improve software performance through the identification of processes that can be run in parallel.

Table 3 demonstrates the critical path analysis of the ETL_modular program (when invoked by Driver_small), with "R" denoting read-only (i.e., shared) locks and "W" denoting read-write (i.e., exclusive) locks required to create or modify data sets or other files.

Operation	Inputs	Outputs
EXTRACT operation	(R) ETL.smallfish	(W) ETL.smallfish_1
TRANSFORM macro	(R) ETL.smallfish_1	(W) ETL.smallfish_2
LOAD macro	(R) ETL.smallfish_2	(W) smallfish_rpt.html

Table 3. Critical Path Analysis for ETL_modular.sas Program

The analysis of operational dependencies reveals that EXTRACT must be completed before TRANSFORM can begin (because TRANSFORM requires read-only access to ETL.smallfish_1, which EXTRACT has exclusively locked) and that TRANSFORM must be completed before LOAD can begin (because LOAD requires read-only access to ETL.SMALLFISH_2, which TRANSFORM has exclusively locked). This type of serialization is both common and necessary in many ETL process flows and demonstrates why conventional ETL processes have are being supplanted by streaming data methodologies and technologies, which lie outside the bounds of this text. However, the analysis also demonstrates no dependencies or interaction between the Smallfish and Medfish data sets, indicating how the Driver_small and Driver_med programs are able to be run concurrently.

Within ETL software, the critical path is commonly prescribed through hardcoded, synchronous logic. For example, the Driver_small program reflects the data in Table 3 with the following statements that are executed in sequence:

```
%extract(lib=etl, dsn=smallfish);

%transform(lib=etl, dsn=smallfish);

%load(lib=etl, dsn=smallfish);
```

Data-driven design, however, can facilitate a more autonomous solution that supports software concurrency (i.e., asynchronous logic and execution). This design can be achieved by constructing a configuration file that lists operations and their respective prerequisites, and by subsequently querying that file to drive autonomous execution of the ETL program flow.

The critical path configuration file is formatted as a comma-separated values (CSV) file and should be saved as critical_path.csv:

```
extract, D:\sas\etl\ETL_modular.sas,
transform, D:\sas\etl\ETL_modular.sas, extract
load, D:\sas\etl\ETL_modular.sas, extract transform
```

The first column represents the process (i.e., macro name); the second column represents the program file in which the process is saved, and; the third column represents a space-delimited list of all respective prerequisite processes (in any order). Note that in this example, because all macros are saved within the same program file (ETL_modular), the file name is repeated across all observations within the configuration file. Also note that the folder location (D:\sas\etl\) will need to be updated to the location previously specified in the &LOC global macro variable.

The configuration file can be ingested with the following code:

```
data etl.critical_path;
    length process $32 importfile $256 prereqs $256;
    infile "&loc.critical_path.csv" truncover dsd delimiter=',';
    input process $ importfile $ prereqs;
run;
```

The next section demonstrates how the driver control table incorporates this configuration file to create variables dynamically.

DRIVER CONTROL TABLE

In the previous ETL_modular program example, separate driver programs were required each time a different data set (e.g., Smallfish, Medfish) needed to be processed. To improve software autonomy and maintainability, a single driver program should instead be constructed that can intelligently determine what processes have already completed, what processes are running, and what processes have completed. At the heart of this functionality lies the control table (&CTRL) that contains the following variables:

- Process – name of the process being executed, which must correspond to the macro name
- Prereqs – space-delimited list of processes (i.e., macros) that must have run successfully before the current process can begin
- Includefile – the path and file name of the SAS program in which the macro is saved
- DSN – data set name, including the SAS library, in LIB.DSN format

- Priority – data set priority, a numeric representation of the priority (on a user-defined scale) in which data sets having higher numbers will be processed before data sets having lower numbers
- DTGStart – date/time representing when a process began
- Stop – date/time representing when a process terminated, regardless of whether the process succeeded or failed
- DTGStatus – status of the process (i.e., SUCCEEDED, IN PROGRESS, TIMEOUT, FAILED), with a missing value representing that a process has not yet started for a specific data set
- JobID – &SYSJOBID value of the SAS instance that executed (or is executing) a particular operation

Each observation in the control table represents a separate data set to be processed through the ETL process flow. In a real-world scenario, the driver program might query a database to identify external tables that should be ingested, or query a specific folder (sometimes termed a *drop zone*) in which text files or data sets had been deposited, after which these unique file names could be added to the driver control table. To simulate this identification of data to be processed (which populates the control table, and which kicks off the ETL program flow), the ETL.DSNs data set is created:

```
data etl.dsns;
    length dsn $32 priority 8;
    infile datalines delimiter=',';
    input dsn priority;
    datalines;
ETL.smallfish,5
ETL.medfish,2
ETL.bigfish,43
;
run;
```

Note that the Priority values instruct the software to process Bigfish first (because 43 is the largest number), followed by Smallfish (5) and Medfish (2). The DSNs data set should still be used in a real-world scenario to drive data processing, but it should be created dynamically from some external process or source rather than created statically here.

In the next section, the CTRL_CREATE macro creates the control table if it does not exist. This enables the control table to be deleted if necessary—for example, if it becomes corrupt. More commonly, this functionality facilitates the smooth creation of the data set when run for the first time or within a new environment. The CTRL_GET_DATA macro subsequently adds the data set names from the DSNs data set if they do not already appear in the control table. This macro ensures that as new external data become available (such as when data sets are added to a drop zone folder), they can be incorporated into the ETL processing queue in real time. Finally, separate macros CTRL_GET_NEXT and CTRL_UPDATE query the control table to determine which process should be run next and update the control table (STGStop and Status variables) when a process terminates, respectively.

With all this bidirectional communication occurring in which processes both read from and write to the control table, and given that multiple, concurrent processes might be attempting to access the control table at one time, file locking is necessary to ensure that file access collisions or other race conditions do not occur. The LOCKITDOWN macro is not described fully in this text, but tests and reserves a read-write (i.e., exclusive) file lock on the control table whenever it is accessed by any process. LOCKITDOWN and other aspects of SAS data set file locking are described and explored in the author's text (Hughes, From a One-Horse to a One-Stoplight Town: A Base SAS Solution to Preventing Data Access Collisions through the Detection and Deployment of Shared and Exclusive File Locks, 2014).

DRIVER PROGRAM RUN IN SERIES

The driver program consists of several reusable macros that create, access, and modify the control table to determine what ETL process should be run next and to track which processes have been completed for specific data sets. The driver program is also responsible for importing processes from external SAS program files, executing those processes, and validating (to some extent) that those processes completed successfully. Driver program macros include the following:

- CTRL_CREATE – This macro creates the &CTRL control table if it does not exist. This enables the control table to be generated in a new environment or regenerated if it becomes corrupted.
- CTRL_GET_DATA – The DSNs control table is queried to determine which data sets should be processed and their respective priorities (i.e., Priority variable). These results are joined with the existent &CTRL control table, which adds processes and data sets not already identified in the control table. This functionality adds tremendous flexibility to the Driver program because it enables new data sets to enter the process queue (i.e., DSNs control table) while Driver is executing.
- CTRL_GET_NEXT – This macro generates four global macro variables that describe the next data set (and respective operation) to be processed: &NEXTNOBS, &NEXTDSN, &NEXTPROCESS, &NEXTINCLUDE respectively represent the number of operations ready to be run, the next data set that should be processed, the next operation that should be executed (on &NEXTDSN), and the program file in which that process (i.e., macro name) is saved. If &NEXTNOBS is zero, this represents that no operations can be run—either because all processes have been executed (or are in the process of being executed) or because prerequisites have not been met for all data sets that remain. When &NEXTNOBS is zero, the &NEXTDSN, &NEXTPROCESS, and &NEXTINCLUDE global macro variables are initialized to Null and the CTRL_GET_NEXT macro exits.
- CTRL_UPDATE – This macro executes immediately before an ETL operation starts (parameterized with STATUS=START) and immediately after an ETL operation terminates (parameterized with STATUS=STOP). When an operation starts, CTRL_UPDATE captures the date/time of the process start as well as the &SYSJOBID of the SAS instance executing the process. When an operation ends, CTRL_UPDATE captures the status as either "SUCCEEDED" or "FAILED." Additional logic, not demonstrated, could further differentiate this status by depicting processes that timed out (i.e., "TIMEOUT) before some user-specified time period.
- DRIVER – This macro loops through the driver control table (e.g., ETL.Ctrl) to process all data sets until the queue is exhausted. At the outset of each loop, the DSNs control table is queried, which external processes (not demonstrated) could have been updating in real time. As ETL processes are executed on various data sets, the driver control table is updated to reflect this progress in real-time.

The Driver program file, which includes the preceding macros, should be saved as Driver.sas and is depicted in Appendix B.

The DRIVER macro—the only macro called directly within the Driver program—is defined as:

```
%macro driver(ctrl= /* LIB.DSN for ctrl table */,
   crit= /* LIB.DSN for critical path data set */,
   dsns= /* LIB.DSN for list of data sets to process */);
```

Once the setup activities have been performed (as indicated in the previous Setup section) and the ETL_modular program has been saved, the following macro invocation calls the DRIVER macro:

```
%let loc=D:\sas\etl\; /* USER MUST CHANGE LOCATION */
libname etl "&loc";
%include "&loc.driver.sas";
%driver(ctrl=etl.ctrl, crit=etl.critical_path, dsns=etl.dsns);
```

When DRIVER executes the first time, it creates the ETL.Ctrl control table, after which the control table is updated within the DRIVER loop, as well as when DRIVER is subsequently called. The EXTRACT macro has not prerequisite processes so it can be executed first. Moreover, the DSNs data set depicts that the Bigfish data set has the highest priority (43), so the EXTRACT macro will be run first to ingest the Bigfish data set. This autonomous selection of processes continues until all EXTRACT, TRANSFORM, and LOAD operations have completed for all data sets.

Table 4 demonstrates a typical execution pattern when the DRIVER macro is run from a single SAS instance. Note that the Prereqs and Includefile variables have been omitted to improve readability of the control table. Note also that the &SYSJOBID is stable across all observations, representing that all operations were executed by the same SAS instance.

Process	Data Set	DSN Priority	Start Time	Stop Time	Status	SYSJOBID
EXTRACT	ETL.BIGFISH	43	20OCT14:09:03:33	20OCT19:09:04:03	SUCCEEDED	8928
EXTRACT`	ETL.MEDFISH	2	20OCT14:09:05:35	20OCT14:09:06:05	SUCCEEDED	8928
EXTRACT	ETL.SMALLFISH	5	20OCT14:09:04:34	20OCT14:09:05:04	SUCCEEDED	8928
LOAD	ETL.BIGFISH	43	20OCT14:09:04:24	20OCT14:09:04:34	SUCCEEDED	8928
LOAD	ETL.MEDFISH	2	20OCT14:09:06:26	20OCT14:09:06:36	SUCCEEDED	8928
LOAD	ETL.SMALLFISH	5	20OCT14:09:05:25	20OCT14:09:05:35	SUCCEEDED	8928
TRANSFORM	ETL.BIGFISH	43	20OCT14:09:04:03	20OCT14:09:04:24	SUCCEEDED	8928
TRANSFORM	ETL.MEDFISH	2	20OCT14:09:06:06	20OCT14:09:06:26	SUCCEEDED	8928
TRANSFORM	ETL.SMALLFISH	5	20OCT14:09:05:05	20OCT14:09:05:25	SUCCEEDED	8928

Table 4. Control Table (ETL.Ctrl) When DRIVER Macro Is Run in Single SAS Session

For each operation (e.g., EXTRACT), the data set names are created in the order in which they are found within the ETL.DSNs data set. However, note that Bigfish was always processed before Smallfish (and Smallfish before Medfish), following the data set priority set by the Priority variable. Thus, not only can new data sets be processed by modifying the ETL.DSNs control table in real time, but those data sets can be effectively prioritized against other data sets queued for processing in the ETL.Ctrl control table. Note that all EXTRACT operations took approximately 30 seconds, all TRANSFORM operations approximately 20 seconds, and all LOAD operations approximately 10 seconds, as specified by the various calls to the SLEEP macro within the ETL_modular SAS program.

At this point, although the control tables and configuration file are directing program flow and the execution of operations, the ETL processes remain serialized because they are run within a single SAS session. For example, the first process (extracting Bigfish) starts at 09:03:33 and the last process (loading Medfish) completes at 09:06:36. As expected, this represents that approximately 180 seconds of processing (30 seconds extracting, 20 seconds transforming, and 10 seconds loading) completed in 183 seconds. The next session demonstrates parallel execution of the DRIVER macro in two SAS sessions.

DRIVER PROGRAM RUN IN PARALLEL

Use of the LOCKITDOWN macro ensures that multiple SAS sessions can confidently access and update the driver control table (i.e., ETL.Ctrl) without fear of file access collisions. To demonstrate the functionally equivalent ETL process run in parallel, the following code can be run in one SAS session:

```
%let loc=D:\sas\etl\; /* USER MUST CHANGE LOCATION */
libname etl "&loc";
%include "&loc.driver.sas";
%driver(ctrl=etl.ctrlparallel, crit=etl.critical_path, dsns=etl.dsns);
```

Note that a new control table (i.e., ETL.Ctrlparallel) has been specified. Because this control table does not exist, it will initially be created, after which the DRIVER macro will begin to select, prioritize, and execute the specified processes autonomously.

Immediately after executing the previous code, the following identical code should be executed within a second SAS session:

```
%let loc=D:\sas\etl\; /* USER MUST CHANGE LOCATION */
libname etl "&loc";
%include "&loc.driver.sas";
%driver(ctrl=etl.ctrlparallel, crit=etl.critical_path, dsns=etl.dsns);
```

The first SAS session again selects the highest prioritized data set (i.e., Bigdata) and runs the EXTRACT operation. However, because the second SAS session cannot perform other processes on Bigdata concurrently (because EXTRACT maintains an exclusive file lock), it selects to run EXTRACT on Smallfish, the next highest priority data set. This autonomous selection of processes and data sets continues until all processes have been run and the queue is empty.

Table 5 demonstrates the ETL.Ctrlparallel control table produced by two concurrent SAS sessions. Note that the Jobid variable depicts the two &SYSJOBID values (8928 and 3592) of these two sessions.

Process	Data Set	DSN Priority	Start Time	Stop Time	Status	SYSJOBID
EXTRACT	ETL.BIGFISH	43	20OCT14:10:14:01	20OCT14:10:14:31	SUCCEEDED	8928
EXTRACT`	ETL.MEDFISH	2	20OCT14:10:15:02	20OCT14:10:15:32	SUCCEEDED	8928
EXTRACT	ETL.SMALLFISH	5	20OCT14:10:14:05	20OCT14:10:14:35	SUCCEEDED	3592
LOAD	ETL.BIGFISH	43	20OCT14:10:14:51	20OCT14:10:15:01	SUCCEEDED	8928
LOAD	ETL.MEDFISH	2	20OCT14:10:15:52	20OCT14:10:16:02	SUCCEEDED	8928
LOAD	ETL.SMALLFISH	5	20OCT14:10:14:55	20OCT14:10:15:05	SUCCEEDED	3592
TRANSFORM	ETL.BIGFISH	43	20OCT14:10:14:31	20OCT14:10:14:51	SUCCEEDED	8928
TRANSFORM	ETL.MEDFISH	2	20OCT14:10:15:32	20OCT14:10:15:52	SUCCEEDED	8928
TRANSFORM	ETL.SMALLFISH	5	20OCT14:10:14:35	20OCT14:10:14:55	SUCCEEDED	3592

Table 5. Control Table (ETL.Ctrlparallel) When DRIVER Macro Is Run in Two SAS Sessions

The three data sets now complete all ETL processing in only 2 minutes 1 second, with the first session (&SYSJOBID of 8928) performing two-thirds of the work. Appendix C demonstrates the log from the first SAS session, showing the recurrent locking and locking of the control table that prevents file access collisions. Although not demonstrated, a third instance of the DRIVER macro could be called from a third SAS session to further reduce the total runtime to approximately one minute.

CONCLUSION

This text demonstrated a traditionally serialized and monolithic ETL program flow and transformed it into a set of modular programs and macros that facilitate parallel ETL processing. This modularity promotes software flexibility and configurability, in that the ETL.DSNs control table and Critical_path.csv configuration file can be modified by end users without the necessity to modify the underlying code. This flexibility enables the five macros within the Driver program to be reused to execute unrelated program flows. Finally, the parallel processing facilitated tremendously increased performance and throughput, as separate SAS sessions are able to autonomously run different processes on different data sets.

REFERENCES

Hughes, T. M. (2014). From a One-Horse to a One-Stoplight Town: A Base SAS Solution to Preventing Data Access Collisions through the Detection and Deployment of Shared and Exclusive File Locks. *Western Users of SAS Software (WUSS).* San Jose, California.

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. LOCKITDOWN MACRO

```
* saved as LOCKITDOWN.SAS;
%macro LOCKITDOWN(lockfile= /* data set in LIBRARY.DATASET or DATASET
    format */,
    sec=5 /* interval in seconds after which data set is re-tested */,
    max=300 /* maximum seconds waited until timeout */,
    type=W /* either (R) or (W) for READ-only or read-WRITE lock */,
    canbemissing=N /* either (Y) or (N), indicating if data set can not
           exist */);
%local i lib tab starttime max loc sleeping;
%global dsid lockerr lockclr;
%let lockerr=;
%let lockclr=;
%let dsid=;
%let type=%upcase(&type);
%if &type=READ %then %let type=R;
%else %if &type=WRITE %then %let type=W;
%let canbemissing=%upcase(&canbemissing);
%if &canbemissing=YES %then %let canbemissing=Y;
%else %if &canbemissing=NO %then %let canbemissing=N;
%let i=1;
* determine whether a libname is included in the filename parameter;
%do %while(%length(%scan(&lockfile,&i,.))>0);
    %let i=%eval(&i+1);
    %end;
%if &i=2 %then %do;
    %let lib=work;
    %let tab=%scan(&lockfile,1,.);
    %end;
%else %if &i=3 %then %do;
    %let lib=%scan(&lockfile,1,.);
    %let tab=%scan(&lockfile,2,.);
    %end;
%else %do;
    %let lockerr=LOCKITDOWN failed because too many levels in file name;
    %goto err;
    %end;
* determine whether libname, data set name, and type are valid and present;
%if %sysfunc(libref(&lib))^=0 %then %do;
    %let lockerr=LOCKITDOWN failed because library %upcase(&lib) not assigned;
    %goto err;
    %end;

%if %sysfunc(exist(&lib..&tab))^=1 %then %do;
    %if &canbemissing=N %then %do;
           %let lockerr=LOCKITDOWN failed because data set %upcase(&lib..&tab)
                         does not exist;
           %goto err;
           %end;
    %else %if &canbemissing=Y %then %do;
           %goto noerr;
           %end;
    %end;
%if &type^=R and &type^=W %then %do;
    %let lockerr=LOCKITDOWN failed because value for TYPE must be either R or W;
    %goto err;
    %end;
%if &canbemissing^=Y and &canbemissing^=N %then %do;
    %let lockerr=LOCKITDOWN failed because CANBEMISSING must be Y or N;
    %goto err;
    %end;
* lock testing;
```

```
%let starttime=%sysfunc(datetime());
%let loc=%sysfunc(pathname(&lib))\&tab..sas7bdat;
filename myfile "&loc";
%do %until(%eval(&dsid>0) or %sysevalf(%sysfunc(datetime())>&starttime+&max));
    %if &type=W %then %do;
          data _null_;
                dsid=fopen('myfile','u');
                call symput('dsid',dsid);
          run;
          %if %eval(&dsid>0) %then %do;
                lock &lib..&tab;
                %if %eval(&syslckrc^=0) %then %do;
                      %put LOCK FAILED;
                      %let dsid=0;
                      %end;
                %end;
          %end;
    %else %if &type=R %then %let dsid=%sysfunc(open(&lib..&tab));
    %if %eval(&dsid^=0) %then %do;
          %if &type=W %then %do;
                %let lockclr=lock &lib..&tab clear;;
                %end;
          %end;
    %else %do;
          %let dsid=0;
          %put SLEEPING;
          %let sleeping=%sysfunc(sleep(&sec,1));
          %end;
    %end;
%if &dsid=0 %then %do;
    %let lockerr=LOCKITDOWN failed after %sysevalf(%sysfunc(datetime())-
          &starttime) seconds to gain %sysfunc(ifc(&type=W,an exclusive,a
          shared)) lock on &lockfile;
    %let lockclr=;
    %end;
%err: %put &lockerr;
%noerr:;
%mend;
```

APPENDIX B. DRIVER SAS PROFRAM FILE

```
* saved as driver.sas;

* initialize path and library;
%let loc=D:\sas\etl\; /* USER MUST CHANGE LOCATION */
libname etl "&loc";
%include "&loc.ETL_modular.sas";
%include "&loc.lockitdown.sas";


%macro ctrl_create(ctrl= /* LIB.DSN for ctrl table */);
%let syscc=0;
%global ctrl_createRC;
%let ctrl_createRC=GENERAL FAILURE;
%if %sysfunc(exist(&ctrl))=0 %then %do;
   data &ctrl;
      length process $32 prereqs $256 includefile $256 dsn $48 priority 8
         dtgstart 8 dtgstop 8 status $32 jobid 8;
      format process $32. prereqs $256. includefile $256. dsn $48. priority 8.0
         dtgstart datetime17. dtgstop datetime17. status $32. jobid 8.;
      label process='Process' prereqs='Prerequisites' includefile='INCLUDE file'
         dsn='Data Set' priority='DSN Priority'
         dtgstart='Start Time' dtgstop='Stop Time' status='Status'
jobid='SYSJOBID';
      if ^missing(dsn);
   run;
   %end;
%if &syscc=0 %then %let ctrl_createRC=;
%mend;



%macro ctrl_get_data(ctrl= /* LIB.DSN for ctrl table */,
   crit= /* LIB.DSN for critical path data set */);
%let syscc=0;
%global ctrl_get_dataRC;
%let ctrl_get_dataRC=GENERAL FAILURE;
proc sql noprint;
   create table &ctrl as
      select * from &ctrl
   union
      select upper(p.process) as process, upper(p.prereqs) as prereqs,
         upper(p.includefile) as includefile, upper(d.dsn) as dsn, d.priority
      from &crit p, etl.dsns d
         left join &ctrl c
            on upper(c.dsn)=upper(d.dsn)
            where missing(c.dsn);
   quit;
%if &syscc=0 %then %let ctrl_get_dataRC=;
%mend;



%macro ctrl_get_next(ctrl= /* LIB.DSN for ctrl table */);
%let syscc=0;
%global ctrl_get_nextRC nextnobs nextdsn nextprocess nextinclude;
%let nextdsn=;
%let nextprocess=;
%let nextinclude=;
%local nextpri;
%let ctrl_get_nextRC=GENERAL FAILURE;
data temp (keep=fmtname type start label);
```

```
   length fmtname $32 type $2 start $80;
   set &ctrl (rename=(status=label));
   fmtname='ctrltab';
   type='c';
   start=catx('*',upcase(dsn),upcase(process));
   label=upcase(label);
run;

proc format cntlin=temp;
run;

data next (drop=i);
   set &ctrl;
   length ready 3;
   where missing(status);
   ready=0;
   if missing(prereqs) then ready=1;
   else do i=1 to countw(prereqs);
      if put(catx('*',dsn,scan(prereqs,i,,'S')),$ctrltab.)='SUCCEEDED'
         then do;
            if i=countw(prereqs) then ready=1;
            end;
      else leave;
      end;
   if ready=0 then delete;
run;

proc sql noprint;
   select count(*) into: nextnobs trimmed
   from next;
   quit;

%if nextnobs>0 %then %do;
   proc sql noprint outobs=1;
      select priority, dsn, process, includefile
         into : nextpri trimmed, : nextdsn trimmed, : nextprocess trimmed, :
nextinclude trimmed
      from next
      order by priority desc;
      quit;
   %end;
%if &syscc=0 %then %let ctrl_get_nextRC=;
%mend;



%macro ctrl_update(ctrl= /* LIB.DSN for ctrl table */,
   dsn= /* LIB.DSN of data set being processed */,
   process= /* process (i.e., macro name) to run */,
   action= /* START or STOP */,
   status= /* [required for STOP only] SUCCEEDED, FAILED, TIMEOUT */);
%let syscc=0;
%global ctrl_updateRC;
%let ctrl_updateRC=GENERAL FAILURE;
data &ctrl (drop=updatetotal);
   set &ctrl end=eof;
   retain updatetotal 0;
   if dsn="%upcase(&dsn)" and process="%upcase(&process)" then do;
      updatetotal+1;
      if upcase(symget('ACTION'))='START' then do;
         dtgstart=datetime();
         status='IN PROGRESS';
         jobid=&sysjobid;
         end;
      if upcase(symget('ACTION'))='STOP' then do;
         dtgstop=datetime();
         status="%upcase(&status)";
         end;
      end;
   if eof then call symputx('updatetotal',strip(put(updatetotal,8.)),'l');
run;
%if &syscc=0 %then %do;
   %if %eval(&updatetotal=1) %then %let ctrl_updateRC=;
   %else %if %eval(&updatetotal=0) %then %let ctrl_updateRC=CTRL TABLE UPDATE
FAILED;
   %else %if %eval(&updatetotal>1) %then %let ctrl_updateRC=TOO MANY CTRL OBS
UPDATED;
   %end;
%mend;



%macro driver(ctrl= /* LIB.DSN for ctrl table */,
   crit= /* LIB.DSN for critical path data set */,
   dsns= /* LIB.DSN for list of data sets to process */);
%let syscc=0;
%local tasks completestatus;
%let tasks=1;
* create control table if it does not exist;
%ctrl_create(ctrl=&ctrl);
* loop until no more processes to run;
%do %while(&tasks^=0 and &syscc=0);
   * look for new data sets to process;
    %lockitdown(lockfile=&ctrl, sec=1, max=30, type=W,
          canbemissing=N);
   %ctrl_get_data(ctrl=&ctrl, crit=&crit);
   * get next process-data combo to run;
   %ctrl_get_next(ctrl=&ctrl);
   %let tasks=&nextnobs;
   * run next process;
   %if &tasks>0 %then %do;
      * update control table with process start;
      %ctrl_update(ctrl=&ctrl, dsn=&nextdsn,
         process=&nextprocess, action=START);
      &lockclr;
      * run next process;
      %&nextprocess(lib=%scan(&nextdsn,1,.), dsn=%scan(&nextdsn,2,.));
      * update control table with process end;
      %if %length(&processRC)=0 %then %let completestatus=SUCCEEDED;
      %else %let completestatus=FAILED;
    %lockitdown(lockfile=&ctrl, sec=1, max=30, type=W,
          canbemissing=N);
      %ctrl_update(ctrl=&ctrl, dsn=&nextdsn,
         process=&nextprocess, action=STOP, status=&completestatus);
      &lockclr;
      %end;
   %else %do;
      &lockclr;
      %end;
   %end;
%mend;
```

APPENDIX C. 1ST SAS SESSION LOG WHEN RUNNING DRIVER IN PARALLEL

```
675   %let loc=D:\sas\etl\; /* USER MUST CHANGE LOCATION */
676   libname etl "&loc";
NOTE: Libref ETL was successfully assigned as follows:
      Engine:        V9
      Physical Name: D:\sas\etl
677   %include "&loc.driver.sas";
NOTE: Libref ETL was successfully assigned as follows:
      Engine:        V9
      Physical Name: D:\sas\etl
1009  %driver(ctrl=etl.ctrlparallel, crit=etl.critical_path, dsns=etl.dsns);

NOTE: Variable process is uninitialized.
NOTE: Variable prereqs is uninitialized.
NOTE: Variable includefile is uninitialized.
NOTE: Variable dsn is uninitialized.
NOTE: Variable priority is uninitialized.
NOTE: Variable dtgstart is uninitialized.
NOTE: Variable dtgstop is uninitialized.
NOTE: Variable status is uninitialized.
NOTE: Variable jobid is uninitialized.
NOTE: The data set ETL.CTRLPARALLEL has 0 observations and 9 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.01 seconds



NOTE: Numeric values have been converted to character values at the places given
by:
      (Line):(Column).
      9:68
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.01 seconds


NOTE: ETL.CTRLPARALLEL.DATA is now locked for exclusive access by you.

WARNING: A table has been extended with null columns to perform the UNION set
operation.
NOTE: The execution of this query involves performing one or more Cartesian product
joins that can
      not be optimized.
WARNING: This CREATE TABLE statement recursively references the target table. A
consequence of
         this is a possible data integrity problem.
NOTE: Table ETL.CTRLPARALLEL created, with 9 rows and 9 columns.

NOTE: PROCEDURE SQL used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds



NOTE: There were 9 observations read from the data set ETL.CTRLPARALLEL.
NOTE: The data set WORK.TEMP has 9 observations and 4 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.01 seconds
```

```
NOTE: Format $CTRLTAB is already on the library WORK.FORMATS.
NOTE: Format $CTRLTAB has been output.
NOTE: PROCEDURE FORMAT used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds

NOTE: There were 9 observations read from the data set WORK.TEMP.


NOTE: There were 9 observations read from the data set ETL.CTRLPARALLEL.
      WHERE MISSING(status);
NOTE: The data set WORK.NEXT has 3 observations and 10 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds


NOTE: PROCEDURE SQL used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds


NOTE: PROCEDURE SQL used (Total process time):
      real time           0.00 seconds
      cpu time            0.01 seconds



NOTE: There were 9 observations read from the data set ETL.CTRLPARALLEL.
NOTE: The data set ETL.CTRLPARALLEL has 9 observations and 9 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.01 seconds


NOTE: ETL.CTRLPARALLEL.DATA is no longer locked by you.

NOTE: There were 477 observations read from the data set ETL.BIGFISH.
NOTE: The data set ETL.BIGFISH_1 has 477 observations and 7 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds



NOTE: Numeric values have been converted to character values at the places given
by:
      (Line):(Column).
      9:68
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds


NOTE: ETL.CTRLPARALLEL.DATA is now locked for exclusive access by you.


NOTE: There were 9 observations read from the data set ETL.CTRLPARALLEL.
NOTE: The data set ETL.CTRLPARALLEL has 9 observations and 9 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.01 seconds
```

```
NOTE: ETL.CTRLPARALLEL.DATA is no longer locked by you.

NOTE: Numeric values have been converted to character values at the places given
by:
      (Line):(Column).
      9:68
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds


NOTE: ETL.CTRLPARALLEL.DATA is now locked for exclusive access by you.

WARNING: A table has been extended with null columns to perform the UNION set
operation.
NOTE: The execution of this query involves performing one or more Cartesian product
joins that can
      not be optimized.
WARNING: This CREATE TABLE statement recursively references the target table. A
consequence of
         this is a possible data integrity problem.
NOTE: Table ETL.CTRLPARALLEL created, with 9 rows and 9 columns.

NOTE: PROCEDURE SQL used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds



NOTE: There were 9 observations read from the data set ETL.CTRLPARALLEL.
NOTE: The data set WORK.TEMP has 9 observations and 4 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.01 seconds


NOTE: Format $CTRLTAB is already on the library WORK.FORMATS.
NOTE: Format $CTRLTAB has been output.
NOTE: PROCEDURE FORMAT used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds

NOTE: There were 9 observations read from the data set WORK.TEMP.


NOTE: There were 7 observations read from the data set ETL.CTRLPARALLEL.
      WHERE MISSING(status);
NOTE: The data set WORK.NEXT has 2 observations and 10 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds


NOTE: PROCEDURE SQL used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds


NOTE: PROCEDURE SQL used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds
```

```
NOTE: There were 9 observations read from the data set ETL.CTRLPARALLEL.
NOTE: The data set ETL.CTRLPARALLEL has 9 observations and 9 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds


NOTE: ETL.CTRLPARALLEL.DATA is no longer locked by you.

NOTE: There were 477 observations read from the data set ETL.BIGFISH_1.
NOTE: The data set ETL.BIGFISH_2 has 477 observations and 8 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds



NOTE: Numeric values have been converted to character values at the places given
by:
      (Line):(Column).
      9:68
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds


NOTE: ETL.CTRLPARALLEL.DATA is now locked for exclusive access by you.


NOTE: There were 9 observations read from the data set ETL.CTRLPARALLEL.
NOTE: The data set ETL.CTRLPARALLEL has 9 observations and 9 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds


NOTE: ETL.CTRLPARALLEL.DATA is no longer locked by you.

NOTE: Numeric values have been converted to character values at the places given
by:
      (Line):(Column).
      9:68
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds


NOTE: ETL.CTRLPARALLEL.DATA is now locked for exclusive access by you.

WARNING: A table has been extended with null columns to perform the UNION set
operation.
NOTE: The execution of this query involves performing one or more Cartesian product
joins that can
      not be optimized.
WARNING: This CREATE TABLE statement recursively references the target table. A
consequence of
         this is a possible data integrity problem.
NOTE: Table ETL.CTRLPARALLEL created, with 9 rows and 9 columns.

NOTE: PROCEDURE SQL used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds



NOTE: There were 9 observations read from the data set ETL.CTRLPARALLEL.
NOTE: The data set WORK.TEMP has 9 observations and 4 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds


NOTE: Format $CTRLTAB is already on the library WORK.FORMATS.
NOTE: Format $CTRLTAB has been output.
NOTE: PROCEDURE FORMAT used (Total process time):
      real time           0.00 seconds
      cpu time            0.01 seconds

NOTE: There were 9 observations read from the data set WORK.TEMP.


NOTE: There were 5 observations read from the data set ETL.CTRLPARALLEL.
      WHERE MISSING(status);
NOTE: The data set WORK.NEXT has 2 observations and 10 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds


NOTE: PROCEDURE SQL used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds


NOTE: PROCEDURE SQL used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds



NOTE: There were 9 observations read from the data set ETL.CTRLPARALLEL.
NOTE: The data set ETL.CTRLPARALLEL has 9 observations and 9 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds


NOTE: ETL.CTRLPARALLEL.DATA is no longer locked by you.
NOTE: Writing HTML Body file: BIGFISH_rpt.html
NOTE: There were 5 observations read from the data set ETL.BIGFISH_2.
NOTE: PROCEDURE REPORT used (Total process time):
      real time           0.06 seconds
      cpu time            0.00 seconds



NOTE: Numeric values have been converted to character values at the places given
by:
      (Line):(Column).
      9:68
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds
```

```
NOTE: ETL.CTRLPARALLEL.DATA is now locked for exclusive access by you.


NOTE: There were 9 observations read from the data set ETL.CTRLPARALLEL.
NOTE: The data set ETL.CTRLPARALLEL has 9 observations and 9 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.01 seconds


NOTE: ETL.CTRLPARALLEL.DATA is no longer locked by you.

NOTE: Numeric values have been converted to character values at the places given
by:
      (Line):(Column).
      9:68
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds


NOTE: ETL.CTRLPARALLEL.DATA is now locked for exclusive access by you.

WARNING: A table has been extended with null columns to perform the UNION set
operation.
NOTE: The execution of this query involves performing one or more Cartesian product
joins that can
      not be optimized.
WARNING: This CREATE TABLE statement recursively references the target table. A
consequence of
         this is a possible data integrity problem.
NOTE: Table ETL.CTRLPARALLEL created, with 9 rows and 9 columns.

NOTE: PROCEDURE SQL used (Total process time):
      real time           0.01 seconds
      cpu time            0.00 seconds



NOTE: There were 9 observations read from the data set ETL.CTRLPARALLEL.
NOTE: The data set WORK.TEMP has 9 observations and 4 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.01 seconds


NOTE: Format $CTRLTAB is already on the library WORK.FORMATS.
NOTE: Format $CTRLTAB has been output.
NOTE: PROCEDURE FORMAT used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds

NOTE: There were 9 observations read from the data set WORK.TEMP.


NOTE: There were 3 observations read from the data set ETL.CTRLPARALLEL.
      WHERE MISSING(status);
NOTE: The data set WORK.NEXT has 1 observations and 10 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.01 seconds
```

```
NOTE: PROCEDURE SQL used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds


NOTE: PROCEDURE SQL used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds



NOTE: There were 9 observations read from the data set ETL.CTRLPARALLEL.
NOTE: The data set ETL.CTRLPARALLEL has 9 observations and 9 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.01 seconds


NOTE: ETL.CTRLPARALLEL.DATA is no longer locked by you.

NOTE: There were 318 observations read from the data set ETL.MEDFISH.
NOTE: The data set ETL.MEDFISH_1 has 318 observations and 7 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds



NOTE: Numeric values have been converted to character values at the places given
by:
      (Line):(Column).
      9:68
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds


NOTE: ETL.CTRLPARALLEL.DATA is now locked for exclusive access by you.


NOTE: There were 9 observations read from the data set ETL.CTRLPARALLEL.
NOTE: The data set ETL.CTRLPARALLEL has 9 observations and 9 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.01 seconds


NOTE: ETL.CTRLPARALLEL.DATA is no longer locked by you.

NOTE: Numeric values have been converted to character values at the places given
by:
      (Line):(Column).
      9:68
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds


NOTE: ETL.CTRLPARALLEL.DATA is now locked for exclusive access by you.

WARNING: A table has been extended with null columns to perform the UNION set
operation.
```

```
NOTE: The execution of this query involves performing one or more Cartesian product
joins that can
      not be optimized.
WARNING: This CREATE TABLE statement recursively references the target table. A
consequence of
         this is a possible data integrity problem.
NOTE: Table ETL.CTRLPARALLEL created, with 9 rows and 9 columns.

NOTE: PROCEDURE SQL used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds



NOTE: There were 9 observations read from the data set ETL.CTRLPARALLEL.
NOTE: The data set WORK.TEMP has 9 observations and 4 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds


NOTE: Format $CTRLTAB is already on the library WORK.FORMATS.
NOTE: Format $CTRLTAB has been output.
NOTE: PROCEDURE FORMAT used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds

NOTE: There were 9 observations read from the data set WORK.TEMP.


NOTE: There were 2 observations read from the data set ETL.CTRLPARALLEL.
      WHERE MISSING(status);
NOTE: The data set WORK.NEXT has 1 observations and 10 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds


NOTE: PROCEDURE SQL used (Total process time):
      real time           0.00 seconds
      cpu time            0.01 seconds


NOTE: PROCEDURE SQL used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds



NOTE: There were 9 observations read from the data set ETL.CTRLPARALLEL.
NOTE: The data set ETL.CTRLPARALLEL has 9 observations and 9 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds


NOTE: ETL.CTRLPARALLEL.DATA is no longer locked by you.

NOTE: There were 318 observations read from the data set ETL.MEDFISH_1.
NOTE: The data set ETL.MEDFISH_2 has 318 observations and 8 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds
```

```
NOTE: Numeric values have been converted to character values at the places given
by:
      (Line):(Column).
      9:68
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds


NOTE: ETL.CTRLPARALLEL.DATA is now locked for exclusive access by you.


NOTE: There were 9 observations read from the data set ETL.CTRLPARALLEL.
NOTE: The data set ETL.CTRLPARALLEL has 9 observations and 9 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.01 seconds


NOTE: ETL.CTRLPARALLEL.DATA is no longer locked by you.

NOTE: Numeric values have been converted to character values at the places given
by:
      (Line):(Column).
      9:68
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds


NOTE: ETL.CTRLPARALLEL.DATA is now locked for exclusive access by you.

WARNING: A table has been extended with null columns to perform the UNION set
operation.
NOTE: The execution of this query involves performing one or more Cartesian product
joins that can
      not be optimized.
WARNING: This CREATE TABLE statement recursively references the target table. A
consequence of
         this is a possible data integrity problem.
NOTE: Table ETL.CTRLPARALLEL created, with 9 rows and 9 columns.

NOTE: PROCEDURE SQL used (Total process time):
      real time           0.01 seconds
      cpu time            0.00 seconds



NOTE: There were 9 observations read from the data set ETL.CTRLPARALLEL.
NOTE: The data set WORK.TEMP has 9 observations and 4 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds


NOTE: Format $CTRLTAB is already on the library WORK.FORMATS.
NOTE: Format $CTRLTAB has been output.
NOTE: PROCEDURE FORMAT used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds
```

```
NOTE: There were 9 observations read from the data set WORK.TEMP.


NOTE: There were 1 observations read from the data set ETL.CTRLPARALLEL.
      WHERE MISSING(status);
NOTE: The data set WORK.NEXT has 1 observations and 10 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.01 seconds


NOTE: PROCEDURE SQL used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds


NOTE: PROCEDURE SQL used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds



NOTE: There were 9 observations read from the data set ETL.CTRLPARALLEL.
NOTE: The data set ETL.CTRLPARALLEL has 9 observations and 9 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.01 seconds


NOTE: ETL.CTRLPARALLEL.DATA is no longer locked by you.
NOTE: Writing HTML Body file: MEDFISH_rpt.html
NOTE: There were 5 observations read from the data set ETL.MEDFISH_2.
NOTE: PROCEDURE REPORT used (Total process time):
      real time           0.07 seconds
      cpu time            0.00 seconds



NOTE: Numeric values have been converted to character values at the places given
by:
      (Line):(Column).
      9:68
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds


NOTE: ETL.CTRLPARALLEL.DATA is now locked for exclusive access by you.


NOTE: There were 9 observations read from the data set ETL.CTRLPARALLEL.
NOTE: The data set ETL.CTRLPARALLEL has 9 observations and 9 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds


NOTE: ETL.CTRLPARALLEL.DATA is no longer locked by you.

NOTE: Numeric values have been converted to character values at the places given
by:
      (Line):(Column).
      9:68
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds


NOTE: ETL.CTRLPARALLEL.DATA is now locked for exclusive access by you.

WARNING: A table has been extended with null columns to perform the UNION set
operation.
NOTE: The execution of this query involves performing one or more Cartesian product
joins that can
      not be optimized.
WARNING: This CREATE TABLE statement recursively references the target table. A
consequence of
         this is a possible data integrity problem.
NOTE: Table ETL.CTRLPARALLEL created, with 9 rows and 9 columns.

NOTE: PROCEDURE SQL used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds




NOTE: There were 9 observations read from the data set ETL.CTRLPARALLEL.
NOTE: The data set WORK.TEMP has 9 observations and 4 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds


NOTE: Format $CTRLTAB is already on the library WORK.FORMATS.
NOTE: Format $CTRLTAB has been output.
NOTE: PROCEDURE FORMAT used (Total process time):
      real time           0.00 seconds
      cpu time            0.01 seconds

NOTE: There were 9 observations read from the data set WORK.TEMP.


NOTE: There were 0 observations read from the data set ETL.CTRLPARALLEL.
      WHERE MISSING(status);
NOTE: The data set WORK.NEXT has 0 observations and 10 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds


NOTE: PROCEDURE SQL used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds


NOTE: No rows were selected.
NOTE: PROCEDURE SQL used (Total process time):
      real time           0.00 seconds
      cpu time            0.04 seconds


NOTE: ETL.CTRLPARALLEL.DATA is no longer locked by you.
```