Paper 8440-2016

A Waze App for Base SAS®: Automatically Routing around Locked Data Sets, Bottleneck Processes, and Other Traffic Congestion on the Data Superhighway

Troy Martin Hughes

ABSTRACT

The Waze application, purchased by Google in 2013, alerts millions of users about traffic congestion, collisions, construction, and other complexities of the road that can stymie motorists' attempts to get from A to B. From jackknifed rigs to jackalope carcasses, roads can be gnarled by gridlock or littered with obstacles that impede traffic flow and efficiency. Waze algorithms automatically reroute users to more efficient routes based on user-reported events as well as historical norms that demonstrate typical road conditions. Extract, transform, load (ETL) infrastructures often represent serialized process flows that can mimic highways, and which can become similarly snarled by locked data sets, slow processes, and other factors that introduce inefficiency. The LOCKITDOWN SAS® macro, introduced at WUSS in 2014, detects and prevents data access collisions that occur when two or more SAS processes or users simultaneously attempt to access the same SAS data set. Moreover, the LOCKANDTRACK macro, introduced at WUSS in 2015, provides real-time tracking of and historical performance metrics for locked data sets through a unified control table, enabling developers to hone processes to optimize efficiency and data throughput. This text demonstrates the implementation of LOCKANDTRACK and its lock performance metrics to create data-driven, fuzzy logic algorithms that preemptively reroute program flow around inaccessible data sets. Thus, rather than needlessly waiting for a data set to become available or a process to complete, the software actually anticipates the wait time based on historical norms, performs other (independent) functions, and returns to the original process when it becomes available.

INTRODUCTION

One of the most common sources of functional failure within production SAS software is the threat of multiple users or processes attempting simultaneously to access the same data set. If a data set is being created or modified, the SAS session maintains an exclusive lock during the process which prevents any other user or process from modifying or even viewing the data set. Thus, anytime an exclusive lock is held on a data set within a shared library accessible by other users or processes, this lock poses a threat to all other potential attempts to access the data set because they will fail. The LOCKITDOWN macro eliminates this risk by testing data set locks before attempted access and is described extensively in a separate text by the author: *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*.[i] The solution demonstrated in this text requires the LOCKITDOWN macro to function.

With implementation of LOCKITDOWN within an exception handling framework, production SAS software can be automated, scheduled, and reliably executed without fear of file access collisions. Because of this newfound reliability and robustness, SAS practitioners are freed to pursue more interesting tasks than log parsing in search of runtime errors or babysitting tenuous software. However, as permanent data sets become more popular (because they can now be reliably accessed without fear of failure), SAS processes may begin to experience increasingly longer wait times to access data sets via the LOCKITDOWN macro. For example, if a critical data set is depended upon by 10 different processes, each of those dependencies will be competing for shared or exclusive file locks which can slow software performance. Thus, data sources that are extremely reliable can become victims of their own success as additional users learn of their reliability and clamor for access.

To facilitate identification of SAS processes that may be unnecessarily using or monopolizing data sets, the LOCKANDTRACK macro was introduced which creates a historical control table that saves all attempted and successful SAS file locks. Through the analysis of *successful* file locks, SAS practitioners can quantify how long specific SAS processes require control over specific SAS data sets, thus demonstrating processes that might benefit from increased performance. And, through the analysis of *failed* file locks, SAS practitioners can determine which processes are blocking other processes from accessing necessary permanent data sets. Armed with this information, SAS practitioners are better equipped to more effectively schedule SAS batch jobs so they don't conflict with other data access requests from other processes. And, where programs are identified that cause substantially long file locks (thus delaying other processes), those programs can be specifically targeted to be made faster and more efficient. The LOCKANDTRACK macro is introduced in a separate text by the author: *Beyond a One-Stoplight Town: A Base SAS® Solution to Preventing Data Access Collisions through the Detection, Deployment, Monitoring, and Optimization of Shared and Exclusive File Locks*[ii]. The solution demonstrated in this text requires the LOCKANDTRACK macro to function.

While the LOCKANDTRACK text introduces the LOCKANDTRACK macro and demonstrates its benefits, it discusses only manual review of the control table to improve software coordination and performance. However, where sufficient data exist to demonstrate historical norms for a specific process on a specific data set, those statistics can be programmatically accessed at the point a file lock is requested to not only alert a process whether a data set is locked (and inaccessible), but also to notify the process the anticipated wait time required for access. For example, when a process executes and attempts to access a data set, it would query the control table, note that the data set had been locked for 20 minutes and (on average) would continue to be locked for another 40 minutes whenever that specific process locked it, and thereafter autonomously decide to perform some other function in lieu of waiting.

THE WAZE APPLICATION

The Waze mobile application provides real-time traffic-related reporting to users to enable them to route around accidents, construction, traffic jams, and other sources of congestion that are reported by users. Other potential hazards such as potholes, broken-down cars, parked emergency vehicles, and other objects in or near roadways also can be reported with only the swipe of a finger. Although Waze provides real-time data for a driver's current direction of travel, it is most useful when a destination is inputted so that Waze can route drivers using the fastest method, and redirecting them if necessary around obstacles. For example, Google Maps uses Waze to determine the fastest routes between two locations and based on shifting traffic, users often receive an audible and visual notification stating "A faster route is now available. Click to accept that route."

In addition to relying on manually-inputted user reports that describe traffic and road conditions, Waze also automatically collects metadata such as user location and driving speed to further validate flow of traffic. When traffic is analyzed with respect to traffic incidents that may have occurred, historical traffic norms can be created that are specific to time of day and day of week and which even account for seasonal variations such as additional school traffic during the schoolyear. These historical metadata enable Google not only to provide real-time traffic updates but also to anticipate future traffic for a specific day and time. And, if Waze were not already awesome enough, Waze users are rewarded with "road candy" for describing traffic conditions and validating the reports of other users, thus facilitating high data quality and integrity.

Data analytic infrastructures often resemble highways with processes attempting to transport data from one form or location to another as quickly as possible. Progress can be impeded, however, where consistent bottlenecks exist such as multiple processes competing for access to the same data set or data sets that are otherwise locked for unreasonable periods of time. Just as Waze relies on historic highway metadata to predict the severity and duration of traffic in specific locations, SAS software also can utilize historic norms of file lock trends to programmatically enable processes to route around data sets that are locked and which are expected to be locked for a considerable amount of time.

DATA ANALYTIC SCENARIO

Consider the scenario in which an ETL process generates a persistent data set PERM.Critical two or three times per week on average. However, smaller transactional updates are made to the data set every four hours, thus two SAS macro processes—CREATE and UPDATE—require exclusive locks on the data set. Several other analytic processes also rely on PERM.Critical but, because they do not modify the data set, only require a shared file lock. However, these analytic processes—represented as the macros FREQ and MEANS—cannot execute while an exclusive lock is maintained by CREATE or UPDATE. Thus, the business problem that exists is that multiple analytic processes may be stuck waiting for an exclusive lock to be released on the same data set. If the analytic processes only have to wait a couple minutes for CREATE or UPDATE to complete, this is rather efficient. However, if the CREATE macro is executing (which takes approximately two hours) and still has substantial time remaining, it would be more efficient to allow the software to instead perform some other unrelated functions than to linger in a busy-waiting spinlock for over an hour.

In this scenario, the first requirement is that all SAS processes that reference the PERM.Critical data set must utilize the LOCKANDTRACK macro and exception handling framework. LOCKANDTRACK gains its integrity from the assurance that all lock attempts—successful and failed—are recorded in the LOCKNTRK.Control control table. For example, if the CREATE macro fails to register that it successfully locks the PERM.Critical data set, other processes (such as the two analytic processes) will not know the average duration of this lock when they attempt to open PERM.Critical but find it is locked by CREATE. Thus, the following CREATE macro definition implements the LOCKANDTRACK macro, as well as the second invocation (with the REMOVE=Y parameter) that terminates the file lock.

```
%macro create();
%lockandtrack(lockfile=perm.critical, sec=1, max=6000, type=W,
canbemissing=NO);
data perm.critical (drop=i);
    length num1 8 char1 $8;
    do i=1 to 1000;
            num1=round(rand('uniform')*100);
            char1=byte(round(rand('uniform')*25)+65); * letters A to Z;
            output;
            end;
run;
%lockandtrack(lockfile=perm.critical, remove=Y);
%mend;
```

A critical process like the CREATE or UPDATE macros would most likely need to wait for PERM.Critical data set to be unlocked if they attempted access and it was locked. For example, if an analytic process like MEANS had a shared lock on PERM.Critical, neither the CREATE nor UPDATE macros would be able to execute because each requires an exclusive lock. But, because of their respective criticality to the overall data infrastructure, both CREATE and UPDATE would need to wait as long as possible. For this reason, they would need to utilize the current LOCKANDTRACK macro while specifying a high MAX parameter, which describes the amount of time the process is willing to wait for a data set to be unlocked.

A less significant macro module like MEANS or FREQ, however, could benefit by not waiting for PERM.Critical if it were locked and were expected to be locked for some extended period of time. The LOCKCHECK macro, demonstrated in Appendix A, is invoked (in lieu of LOCKANDTRACK) and thus additionally queries historical file lock statistics to dynamically drive program flow

```
%lockcheck(dsn=perm.critical, sec=1 ,max=1200, type=R, canbemissing=NO)
%if &lockcheckRC=YES %then %do;
    %freq();
    %lockcheck(lockfile=perm.critical, remove=Y);
    %end;
* do other stuff if lock not available;
```

When LOCKCHECK is invoked, it queries the LOCKNTRK.Control control table, determines that the CREATE process currently has the data set PERM.Critical locked, that it has had the data set locked for 20 minutes, and that on average when the CREATE process locks this data set it is locked for 60 minutes. Thus, the exclusive lock on the data set will be expected to last for another 40 minutes, although the MAX parameter specified by the LOCKCHECK macro invocation is 1200 (or 20 minutes). Because the anticipated wait time exceeds the MAX parameter, the LOCKCHECK return code (the global macro variable &LOCKCHECKRC) is returned as “NO” so that the FREQ module does not execute. However, if the CREATE process instead had already been running for 55 minutes and thus only had five minutes (on average) remaining, the LOCKCHECK macro instead would have entered a busy-waiting spinlock in which it tested the availability of the data set PERM.Critical every second for 20 minutes or until it gained access to the data set.

LOCKCHECK MACRO

Despite having additional functionality, the LOCKCHECK macro invocation is identical to the LOCKANDTRACK invocation to facilitate extensibility from one method to the other.

```
%macro LOCKCHECK(lockfile= /* data set in LIB.DSN or DSN format */,
    sec=5 /* interval in seconds for retesting file lock */,
    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), if DSN can not exist */,
    remove= /* (Y) or (YES) to release a current lock */);
```

- LOCKFILE – Data sets typically only need to be locked when they reside in permanent SAS libraries that are accessible to other users and to concurrent SAS sessions. Notwithstanding, LOCKCHECK can be used on data sets in the WORK library by omitting the library token in the LOCKFILE parameter but, in practice, this should never be done.
- SEC – If a lock cannot be achieved, the SAS session will sleep a number of seconds before reattempting access. Thus, this parameter also acts to prioritize sessions, some of which simultaneously may be waiting for the same locked data set. For example, a program with a 1 second sleep interval will be approximately ten times more likely to achieve a lock than a program simultaneously attempting access but using a 10 second sleep interval. In this manner, the most critical processes can be prioritized over less significant ones by reducing their SEC parameter.
- MAX – After an unacceptable maximum delay, the *event*—a locked file—becomes an *exception* and the macro times out, returning a corresponding error return code. When transactional or standardized data sets are being processed, process time estimation typically is highly correlated with file size and thus can be estimated with some degree of certainty. If multiple processes are anticipated to be attempting access to the same data set, however, the MAX parameter may need to be increased to account for a process that may have to wait for several other processes first to obtain and then to release locks in sequence on the same data set.
- TYPE – The appropriate lock type—either exclusive (i.e., read-write) or shared (i.e., read-only)—must be chosen contextually based on the required action. Failure to select the correct lock type could lead to a data set obtaining only read-only access and producing a runtime error if the subsequent process actually required read-write access. Or, conversely, requesting a read-write lock when only a read-only lock is required unnecessarily locks out other processes from the data set.
- CANBEMISSING – This parameter indicates whether a data set must exist. For example, often in dynamic processes, a data set may be created during the first iteration, and subsequently modified repeatedly thereafter, in which case it would not exist during the initial iteration. If the data set conversely can never be missing, LOCKCHECK first determines file existence and, if the data set does not exist, halts and produces a return code indicating this error.
- REMOVE – When this optional parameter is included, it indicates that a lock is being released rather than established. Moreover, when REMOVE is included, only the LOCKFILE parameter must also be included in the macro invocation.

CONCLUSION

The LOCKCHECK macro includes all functionality of the LOCKANDTRACK macro and in fact requires invocation of both LOCKANDTRACK and LOCKITDOWN macros, previously published as separate texts by the author. LOCKCHECK additionally determines what process currently has locked a data set in question, statistically analyzes historical data to determine the expected lock duration, and thus is able to determine if the data set will be available within a parameterized amount of time. LOCKCHECK facilitates faster processing by allowing processes to predict anticipated wait times for data sets and, if program flow is sufficiently modular, to enable program flow to shift to other processes that can access their required data sets.

REFERENCES

[i] Hughes, Troy Martin. 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).
[ii] Hughes, Troy Martin. 2015. Beyond a One-Stoplight Town: A Base SAS® Solution to Preventing Data Access Collisions through the Detection, Deployment, Monitoring, and Optimization of Shared and Exclusive File Locks. Western Users of SAS Software (WUSS).

CONTACT INFORMATION

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

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