Moonshot Trading

Performance Report

April 14 2018

SEng 468 : Spring 2018

Group G12

Heather Cape V00795197
Geoff Lorne V00802043
Tanner Zinck V00801442

Table of Contents

1.0 Introduction

Our first and foremost goal for the trading system we developed was for it to be highly performant. Performance was on our minds during every stage of the development process, including architecture design, technology choices, and implementation. Throughout the development of the system, we absolutely did not compromise business requirements for performance whatsoever, though we did focus most of our developer effort on attaining more performance.

Our final system achieved 12154 transactions per second, with a cost of $433.70 dollars due to quotes requested from the legacy quote server. Our system meets all business requirements as set out by DayTrading Inc., while also providing the capabilities to scale to a much larger number of users in a real world setting.

2.0 Assumptions

For the purposes of this project and this report, we have assumed that transactions which result in errors are still valid transactions, even if they make no change to the overall state of the system by being run, and that any log file verifiable by the provided verification service is considered an “OK” run. However, it is worth noting, that while we operated under these assumptions for the development of the system, we did not exploit them in any way, and made every attempt possible to produce a system that does not cut any corners, while providing the utmost performance possible.

3.0 Performance Improvements

Below in Figure 1 is the TPS results of the verified files for the milestones over the semester. The orange bars represent the difference between the maximum and minimum runs submitted on the same day and will be discussed later.

Overall, the transaction speeds appear to increase exponentially throughout the semester. Part of the reason for this is that the faster the system is processing commands, the less frequently the caches quotes will expire, and the less likely they will need to be refreshed by the slow legacy server. For example, the first successful 1000 user run took over 15 minutes to complete at 967 TPS. This means that a single quote may need to be reset 15+ times, whereas the best 1000 user run only took approximately 1 minute 30 seconds. This means the quotes would only need to be refreshed once or twice in the entire run.

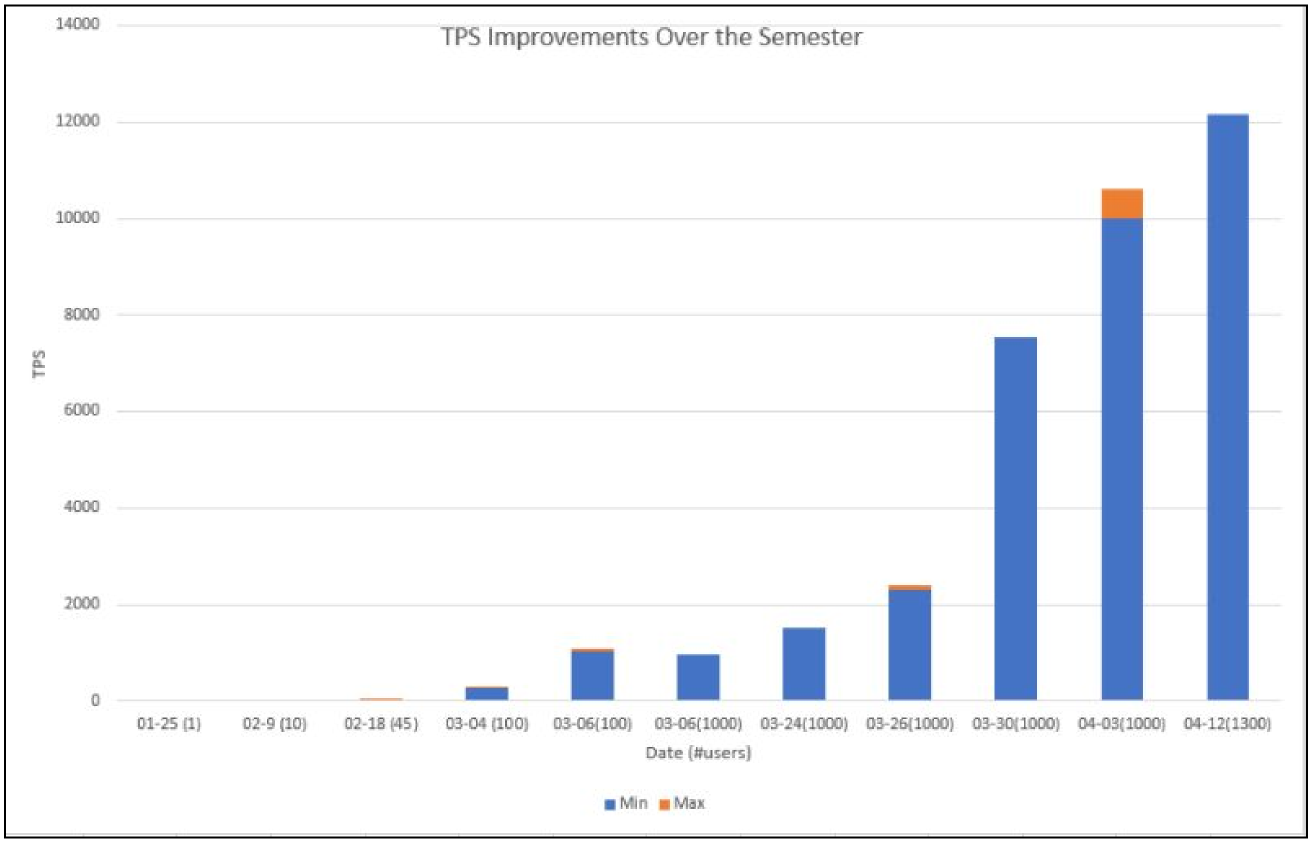


Figure 1: Overall TPS for the semester

3.1 One to One Hundred Users

As visible in the graph above, the later TPS dwarf out the earlier ones making it harder to see the improvements from that time. Figure 2 shows the early improvements more clearly.

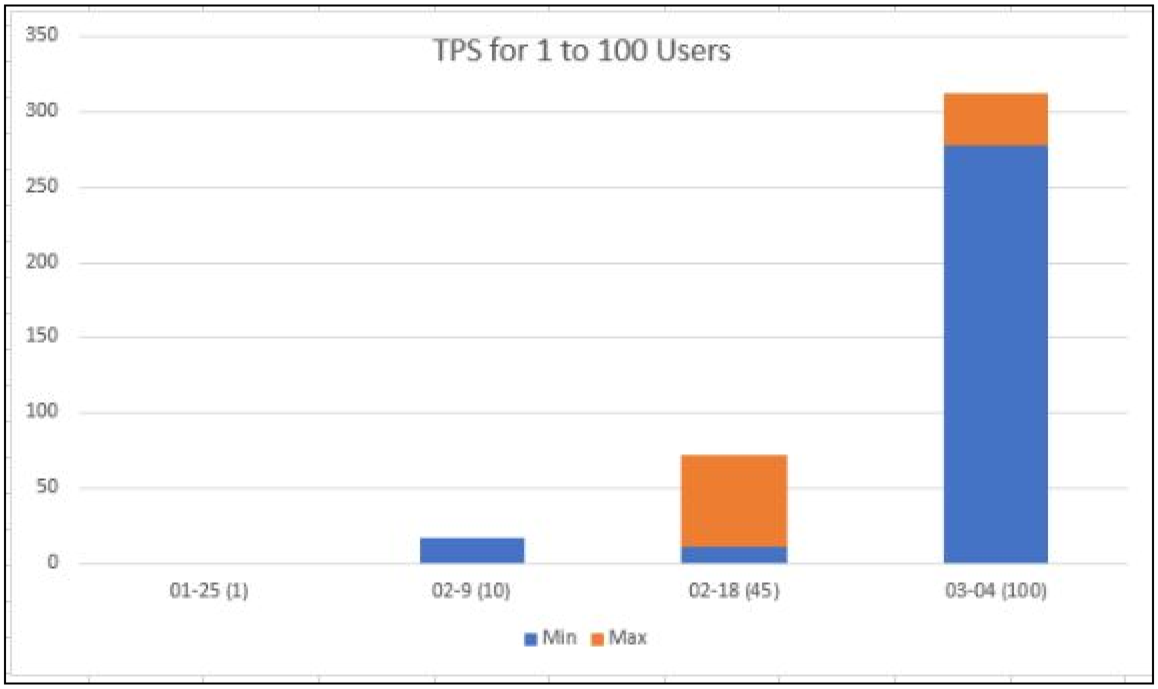


Figure 2: Improvements from the first 1 user run to the first 100 user run

The verification site does not calculate the actual TPS for the one user test as it is rather trivial, but based on the amount of time it took it was actually approximately 19 TPS.

Very early on the main focus was getting a system up that filled the basic business requirements, including all the commands and all the presumed pieces. This also leaked a bit into 10 user test where we were still making design decisions about how to handle triggers. Unlike some other groups (presumably, from anecdotal evidence), we had databases rather than doing everything in memory, a website demo, and separate servers (web, transaction, audit, quote) rather than one monolithic one by the first milestone. This helped us immensely in getting to work on performance optimization early on.

The problems faced early on were mostly caused by unfamiliarity with Go and the way it does concurrent processes with goroutines. The first 45 user run we submitted was a mere 11 TPS, below some other groups in the class. It was here that we realized our workload generator was not actually creating parallel processes for each user; a script was used to split up the large workload file into one file per user, but the generator was not loading them into parallel processes. This was fairly difficult bottleneck to find because the workload generator acts as a part outside the system, that presumably would not be considered part of the problem.

A quote cache was also added between the 10 user run and the 45 user run, as it was something that was going to be necessary but could not be finished in time for the 10 user milestone. These two fixes boosted the performance from 11 to 72 TPS for the 45 user file.

At this point a new emergent behaviour was discovered; the concurrent processes were running fast enough that some of our map data types were being concurrently written to, causing the transaction server to crash. This was fixed by using an externally created type called SyncMap to which is concurrency-safe.

Unfortunately, by the time this bug was fixed we only had a couple days to hit the 100 user milestone. It was apart of one of the earlier milestones to distributed across multiple computers. Using Docker utilization statistics we determined that the audit database would benefit from being moved to a seperate computer. The next problem was that the audit server was running out of connections. An asynchronous queue from the transaction server to the audit server was implemented, along with some OS and PSQL config tweaking. The result of this was that all the transactions would finish processing, but the audit server would take far too long (over 30 minutes) to commit all the transactions to the database.

A batch commit service was implemented, where the audit server takes 20 messages at a time and inserts them on one connections. Buffered Golang channels also provided a way to limit the number of open connections to the database to 80, ensuring that the database would never crash due to open connection limits. We did several runs by modifying these two variables which showed no noticeable improvement, this signified that we had solved the current bottleneck and that the new one was elsewhere. Overall, tackling this bottleneck improved our performance from 72TPS at 45 user to 312 at 100 users.

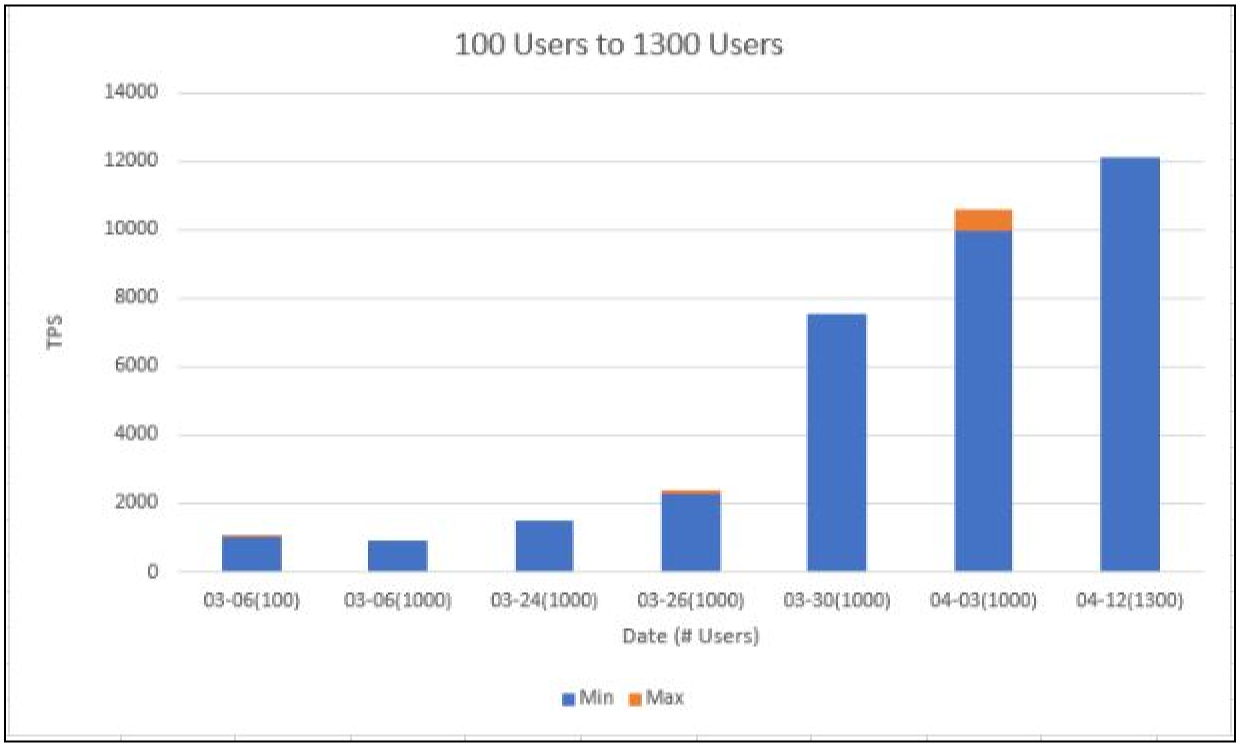
3.2 One Hundred to 1300 Users

At this point the limits were on resources from each computer. We distributed out each server and database to their own computers in order to monitor their maximum CPU utilization. The databases were always maxed out at 350%+ (out of 400%, 100% per core). The transaction and web servers had high utilizations of 150%. It was suspected that the Golang processes could not run well on multicore, and that this was a limit for the servers. The audit and quote servers were fairly underutilized at 80% and 40% respectively.

Once everything was distributed we managed to break the 1000 TPS threshold. Adding a second transaction server with transaction database and splitting the requests from the web server increased the TPS from 1507 to 2403. At this point many changes were made without doing a full run. We realized that we were preparing the SQL statements each time before commiting the queries to the databases; prepare statements are mean to be reused for the same query text so we made that change. The write-through cache was implemented on each transaction server in order to minimize the amount of talking to the resource-intensive DBs. We also configured the databases by turning synchronous commits off and rate limiting all of our connections with Golang buffered channels to avoid crashes. With those changes, and a third transaction server with database, the resulting TPS was a tripled 7306.

At this point two more transaction servers were added for a total of 5. The web server would always crash under the TCP connection load here, even with rate limiting in the workload generator and web server. This meant that if another web server was added, it might be able to handle the load of the increased connections. Once the new configuration was set up --multiple workload generators processing through the HAProxy to two web servers, the result was 10002 TPS. Adding a third web server at this point only lead to a 612 TPS increase, so we suspected that 2 was the current asymptotic limit for web servers.

For the final workload another transaction server was added for a total of 6. At this point there was one service on every (working) computer in the lab. More could be added manually by doubling up the lesser intensive services such as the quote server and workload generators. This also runs the risk of maxing out connections again however. The final TPS was 12154.



3.3 Margin of Error

This section will briefly touch on the margin of error. We should have been more diligent in recording the completion times for multiple runs. Near the end, for the 1000 user runs and above we would run the test multiple times, but only submit the ones that were faster for verification in order to save ourselves time. This time was determined by a print out from the workload generator(s) once they finished sending all messages. We expect that the margin of error comes from other processes running on the computers, and memory availability.

Sometimes the runs ran slower after consecutive trials, and restarting the slow computers alleviated issues.

The average difference between minimum and maximum runs of the same setup was 4.7%.

3.4 Quote Performance

Unfortunately the quote data is not as rich as desired. This is because the class experienced troubles with getting validated log files due to the cryptokeys from the quote server failing. First it was believed to be caused by logging duplicate keys, followed by decryption errors that were never solved, but presumably had to do with underlying encoding issues with Golang. Therefore for some verified files we had to remove all the quote logging in order to make a submission. The results are in Figure 4 below.

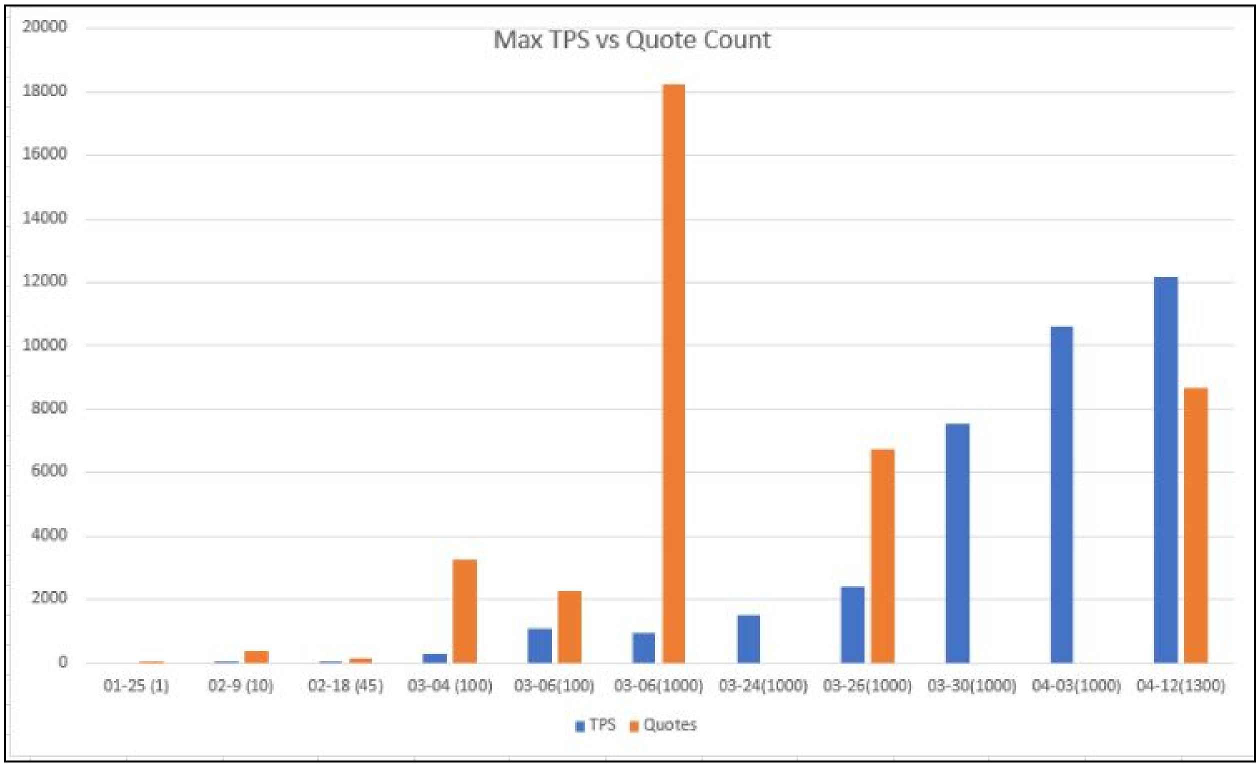


Figure 4: Quotes related to performance

The most notable changes to the quote values are at the 02-09 10 user run where 410 quotes were logged. Another run of the 10 users was done with some performance improvements and added quote cache and the quote count dropped to 63. From there it is interesting to observe the relationship between quotes and TPS. As discussed above, the faster the system can process commands, the more reuse that can happen from the cached quotes.

This data is important in cases like 03-06 100 user vs 1000 user. The resulting TPS was the same, but the quote count (and time to complete the file) show that there is not a lot of reuse happening, so the system performance would need to be improved to make the cache useful.

A graph like this would also be useful if there were multiple different user files of the same size, it could help highlight if one user file has a bigger or smaller spread of different stock names than another.

3.5 Response Modelling

In the lab we used a script in Python to repeatedly hit the legacy quote server and collect the response times. After graphing the result it became clear that the response times were bucketed into distributions of less than 10ms, and around 1, 2, 3, and 4 seconds. This with this information a timeout was set on the quote server connections. If the time to receive a request was over 100ms it would be abandoned and retried.

Using this information relies on the assumption that the legacy server distribution would not change, but it easily could. At the very least we could rerun the script and graph generation and change the timeout value in the configuration file if necessary. In the future, leveraging more than one port to the legacy quote server and modelling the response times dynamically would make for a more realistic system.

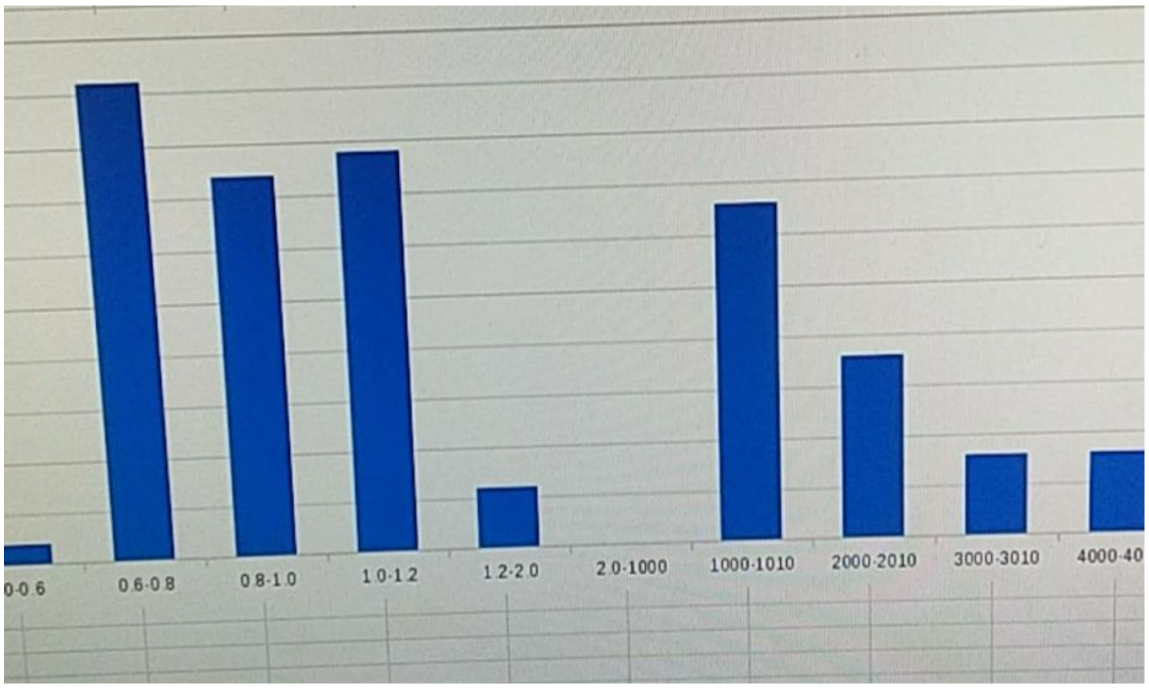


Figure 5: Histogram of response times from legacy server

4.0 Future Considerations

This section of the report will outline some of the future work we would like to do to increase performance of the system. Other performance tweaks are included in the the architecture document in the form of architectural improvements.

4.1 Dynamic Modelling of Legacy Quote Server

To optimize our utilization of the legacy quote server, we performed a very simple statistical analysis to attempt to optimize the timeout before re-requesting a new quote (discussed in Performance documentation). This works well, but only for as long as the legacy quote server stays in the same mode of operation as it was analysed in. Based on our observations of very different behaviours over time, it seems that the legacy quote server is a non-ergodic system, so we would like to implement a dynamic modelling of the system to constantly ensure we are using an optimal timeout length for the current operation mode when requesting quotes.

4.2 Quote Prefetching

Although the data layer of our system has tended to be the bottleneck, the legacy quote server is still the second most time-costly portion of our system, with the 1300 user workload file requiring 8674 request. Quote prefetching is something we would like to implement. The basic idea is that since many stocks are used by more than one user, it would be advantageous to request a fresh quote for each stock (or at least the most popular ones) as soon as the current quote expires. Our modular architecture would allow us to simply modify the legacy quote server wrapper, which provides a measure of transparency to the rest of the system in its interactions with the legacy quote server. We are already using the Redis expire attribute to remove the quotes from the cache, the next step would be to create a publish-subscribe messaging on the expiry event.

4.3 Code Profiling

Given that with our current architecture, we are operating with most system components fairly low in terms of resource utilization (application server far under %50), to achieve higher performance, we will need to start looking more towards the actual code handling each transaction to optimize computation time and lower the asymptotic bound for processing each user command. A good starting point for this process would be to use the official Go “pprof” package, which provides excellent insight into which code is taking the most time, and other useful information. The reason this type of optimization was not worked on during the development of the system thus far is because while this could potentially gain us a speedup in the neighbourhood of ~0 - 15 percent, the architectural optimizations we instead chose to implement effectively gave us a 1000 percent performance increase.

4.4 Variable Tuning

There are a few configuration variables in the code that we did not get a change to fully test for improvements. They are: the number of rows that are bulk inserted at onces into the audit server, the number of connections rate limited to the PSQL, the number of connections rate limited from the web server to the transaction servers, and the optimal time to clear out old buy and sells from the in memory stacks.

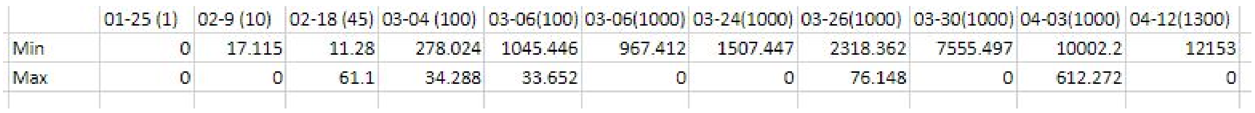
4.5 Statistics

Obtaining more information about the system through statistics gathering could show us locations for improvement. One idea is to record the number of cache hits from the quote server, and compare that to the memory trade off to see if the value is still there. Also, we wanted to run tests on the individual commands to see which commands were taking the longest to run. By the time we wanted to start optimizing this further, we realized that it would involve a much bigger architecture change that any benefit would be at the time. One simple simulation was run with the one user workload file on the final version of the system. Every command came through at less than one millisecond. We expected the commits to the transaction databases to be at least in the range of 100 milliseconds each, so the hypothesis is that they are being asynchronously committed. The information at this scale was not too useful to us, so in the future we would like to develop a better statistics gathering platform

Add	0.10ms
Buy	0.018ms
Quote (uncached)	0.15ms
Quote (cached)	0.015ms
Sell	0.013ms
Commit Buy	0.020ms
Commit Sell	0.016ms
Cancel Buy	0.015ms
Cancel Sell	0.014ms
Set Buy Amount	0.013ms
Set Buy Trigger	None
Set Sell Amount	0.012ms
Set Sell Trigger	0.018ms
Display Summary	0.030ms

Appendix

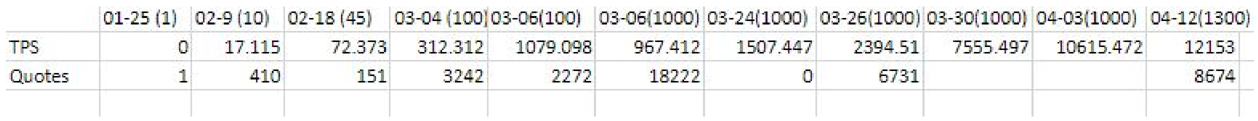
Raw data for TPS performance



	01-25 (1)	02-9 (10)	02-18 (45)	03-04 (100)	03-06(100)	03-06(1000)	03-24(1000)	03-26(1000)	03-30(1000)	04-03(1000)	04-12(1300)
Min	0	17.115	11.28	278.024	1045.446	967.412	1507.447	2318.362	7555.497	10002.2	12153
Max	0	0	61.1	34.288	33.652	0	0	76.148	0	612.272	0

(max is actually the difference between the max run and min run in order to do graph visualization)

Raw data for TPS and Quotes



	01-25 (1)	02-9 (10)	02-18 (45)	03-04 (100	03-06(100)	03-06(1000)	03-24(1000)	03-26(1000)	03-30(1000)	04-03(1000)	04-12(1300)
TPS	0	17.115	72.373	312.312	1079.098	967.412	1507.447	2394.51	7555.497	10615.472	12153
Quotes	1	410	151	3242	2272	18222	0	6731			8674