Table of Contents

Introduction

A supermarket chain, Tigle, is embarking on a rapid and massive expansion in their business. They're in need of a solution to integrate these new stores into their existing business system and hired us to help them build the new system.

For the second part of the contract, we extended the server and connected it to the database. The server will take the client's Purchase POST requests and save them in the database. Server now uses a database connection pool (DBCP) implementing Java Database Connectivity (JDBC) API. HikariCP JDBC connection pooling framework replaces the basic Apache DBCP library to resolve any database deadlocks that can occur.

Data Access Objects (DAO) are used as a middle layer between the server and the database, allowing the insertion of the data in the format we specify.

Business is improving at the Tigle, so we also modified the client to send 300 Purchase POST requests per hour instead of the initial 60.

The implementation of the server can be found in the Github repository below.

Git repo: https://github.com/hjyou07/Distributed_Systems_Project/tree/A2

Server Design

DataSource

DataSource object is used to obtain a connection to the database and allows us to perform JDBC related actions. Initially, the BasicDataSource from Apache was used, but it was replaced with HikariCP's HikariDataSource after the performance review. HikariCP, as mentioned in the introduction, is a fast, simple and "zero-overhead" JDBC connection pool framework.

This is where all the configuration needed to connect to a relational database happens. The hostname, port, name of the schema, username and the password to the database is configured here, as well as some other parameters such as maximum pool size and maximum life-time of a thread. Parameters that contain sensitive information, such as username and password, are read in from the configuration file of the servlet container – Tomcat.

PurchaseDao

Now that we have defined the data source, we can use it to obtain a connection to the database and execute queries. PurchaseDao acts as a buffer between SuperMarketServlet and DataSource and make this happen.

It has a single public method, createPurchaseInDB(), that takes in a Purchase POJO object constructed in the server from the client's purchase request and writes to the database. Our database is a RDS instance deployed in AWS with MySQL engine.

Purchase

Purchase is a POJO class that represents the client's purchase as they will be stored in the database. This is different than the Purchase model defined by the client's Swagger API, as it is optimized for database insertion operation. This POJO saves the store ID, customer ID, the purchase date, and the items purchased as an entire json string from the request body.

SupermarketServlet

This class represents the server and processes the client requests with various loads. The servlet is based on HTTP Protocol, and has two public methods – doGet() and doPost(). Both methods do some parameter validation in the request URL. In doGet() method, we aren't doing much other than validation itself, and return 200 OK HTTP response if valid. In doPost() method, we create the Purchase POJO with the request body then call createPurchaseinDB() with the instantiated PurchaseDAO object, writing to the database.

Database can be a bottleneck in our design as it has to support insert heavy workload. As we increased the number of purchases per hour from 60 to 300, and operating hundreds of stores, the number of POST requests are in folds of few hundred thousand to few million. This is why the server uses a rather simplified Purchase POJO instead of the accurate representation of the Purchase model, and HikariCP framework over Apache DBCP library.

Statistical Analysis

Single Server Tests

Single server initially had failed to process all the client's requests with Apache's basic dbcp library. It didn't seem like the server failure was coming from the server itself, but rather the database. After the switch to HikariCP, single server was sufficient enough to handle the increased client request on its own. This confirmed that the bottleneck present was in the database. The server didn't fail but took too long to handle the requests at 512 threads, that the process itself was terminated before it finished executing.

The mean response time and throughput is plotted against the number of threads in Figure 1. It is observed that the mean response time linearly increases with increasing number of threads – note that the number of threads is increasing exponentially, so the apparent exponential increase in response time is in fact linear.

The throughput peaks at 128 threads then decreases past that point. This indicates that the server and the database have the capacity to handle and indeed benefits from multithreaded requests, but the overhead exceeds the benefits, and the process becomes less efficient past 128 threads.

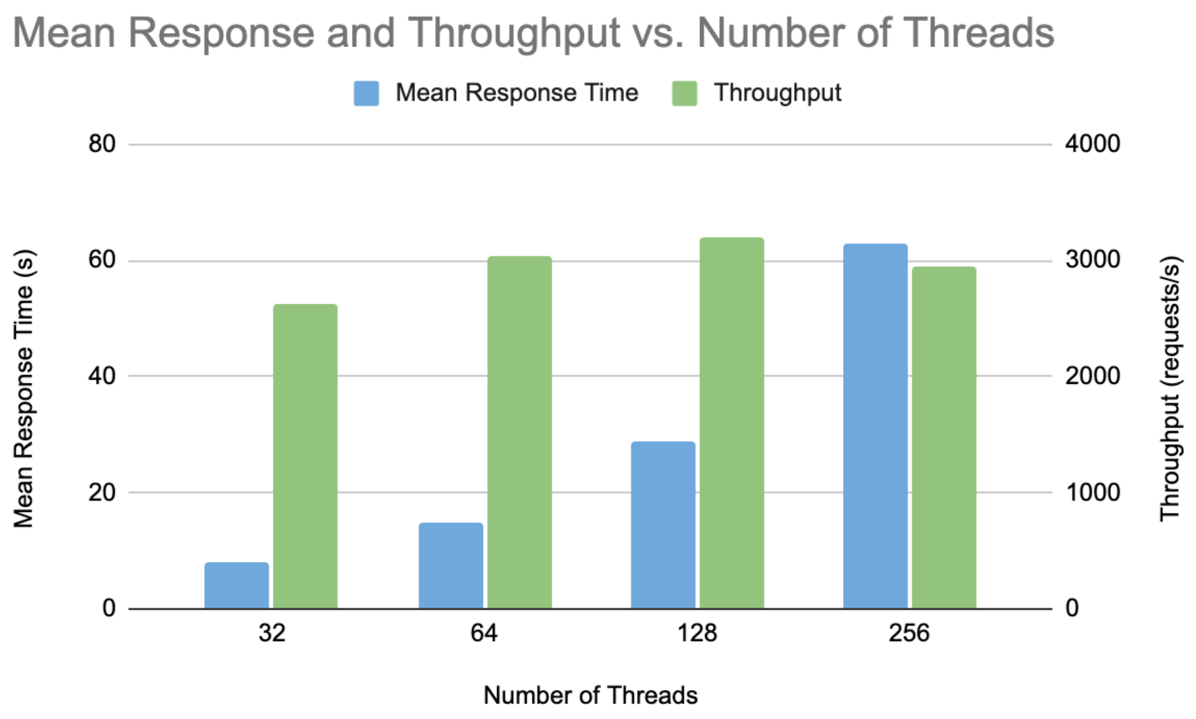


Figure 1. Single Server Tests – Mean Response and Throughput vs. Number of Threads

Load Balancer Tests

Load balancer was configured to distribute the client's requests to 4 replicated servers. The client's requests was directed to the load balancer instead of a single server instance, and the load balancer was responsible for the routing of the requests.

The result indicated a significant performance improvement. The rate at which the minimum response time increases has dropped, that even 1024 threads could be executed without having to be terminated.

The mean response time and throughput against the number of threads is shown in Figure 2. The throughput here peaks at 512 threads then decreases. Compared to that of a single server which peaked at 128 threads, it is seen that the load balancer is able to effectively decrease the overhead on a single server. However, we still have a single instance of the database, and despite using an extremely light JDBC framework, it remains a bottleneck.

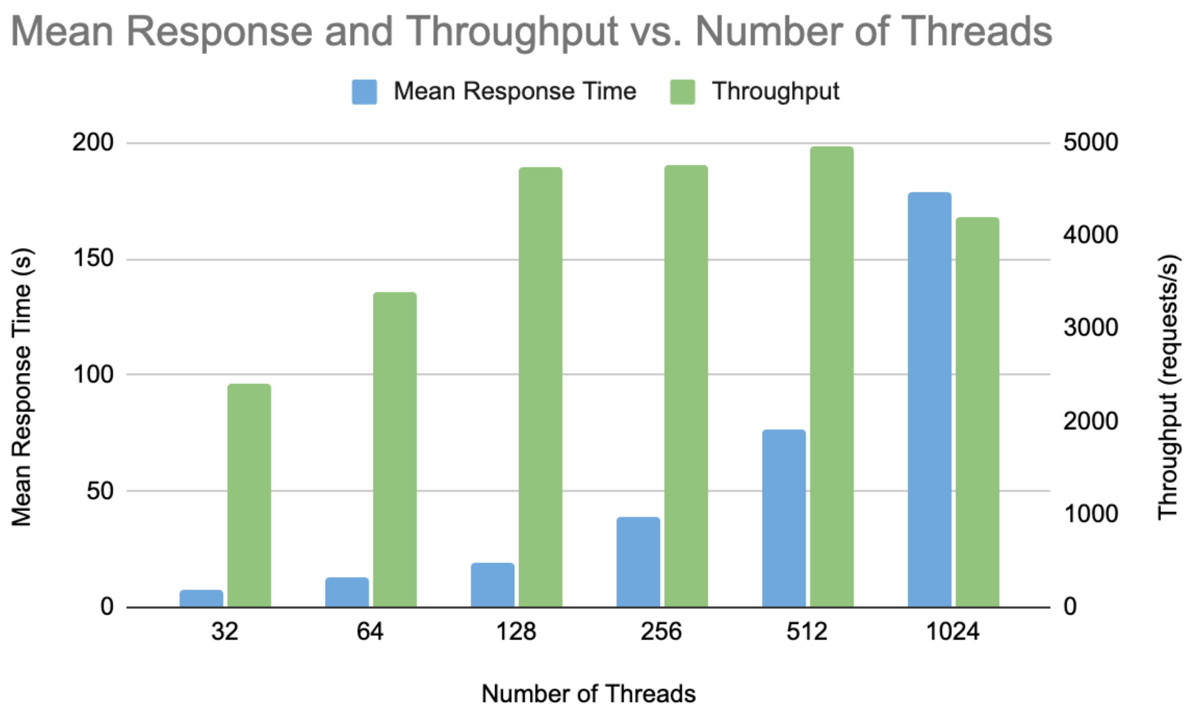


Figure 2. Load Balanced Server Tests – Mean Response and Throughput vs. Number of Threads

Single Server vs Load Balancer

Table 1. Single Server vs. Load Balancer Performance Comparison

Wall Time Comparison (Seconds)				Throughput Comparison (requests/second)			
Threads	Single	Load Balanced	% Difference	Threads	Single	Load Balanced	% Difference
32	33	36	9.09	32	2618	2400	-8.33
64	57	51	-10.53	64	3031	3388	11.78
128	108	73	-32.41	128	3200	4734	47.94
256	234	145	-38.03	256	2953	4766	61.40
512	-	279	-	512	-	4954	-
1024	-	658	-	1024	-	4201	-

Table 1. compares the wall time and the throughput of a single server and load balanced servers against different number of threads, and also shows the percent difference between the two. An interesting point is that for 32 threads, load balanced servers have a longer wall time(hence less throughput) than that of a single server.

In this case, the overhead of adding the load balancer and an extra "hop" for the requests doesn't exceed the benefit of taking the burden off a single server. This newly introduced overhead becomes insignificant as more threads are ran, and we can observe that the performance improves to a greater degree with a greater number of threads.

The percent difference between the load balanced servers and a single server is the highest at 38% and 61% with 256 threads for wall-time and throughput respectively. Sixty-one percent difference in throughput is remarkable.

Refer to Figure 3. and Figure 4. for a visual representation of the comparison.

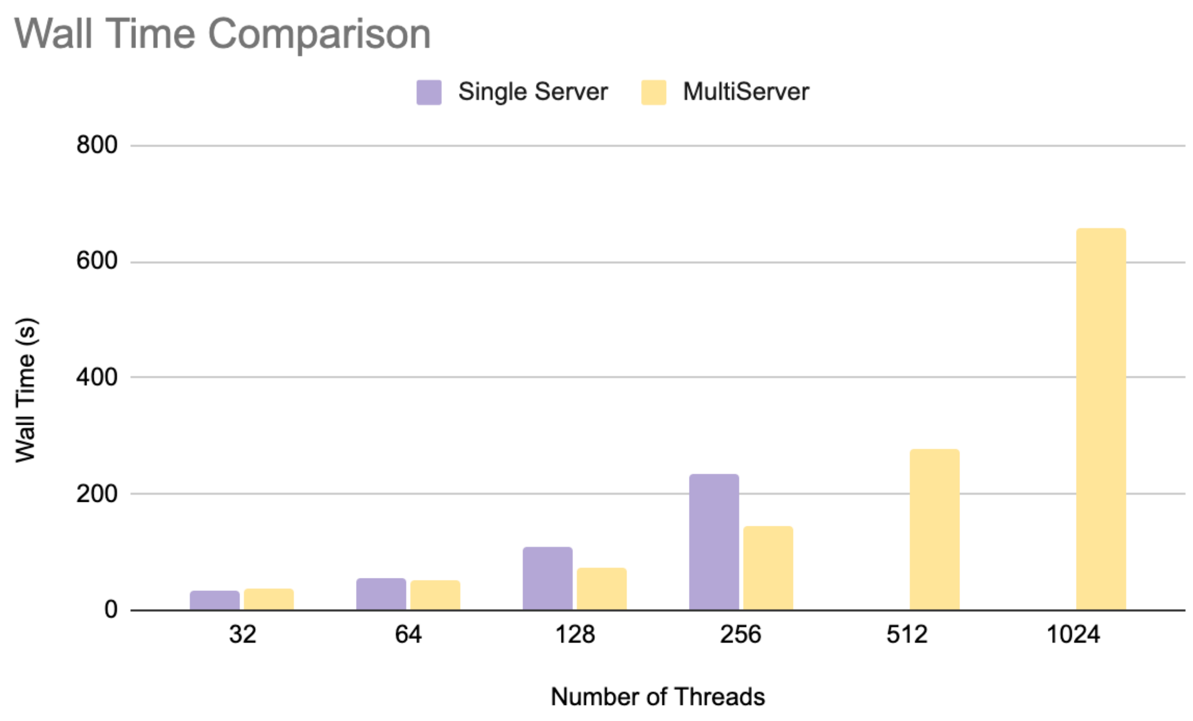


Figure 3. Single Server vs. Load Balanced Server Wall Time Comparison

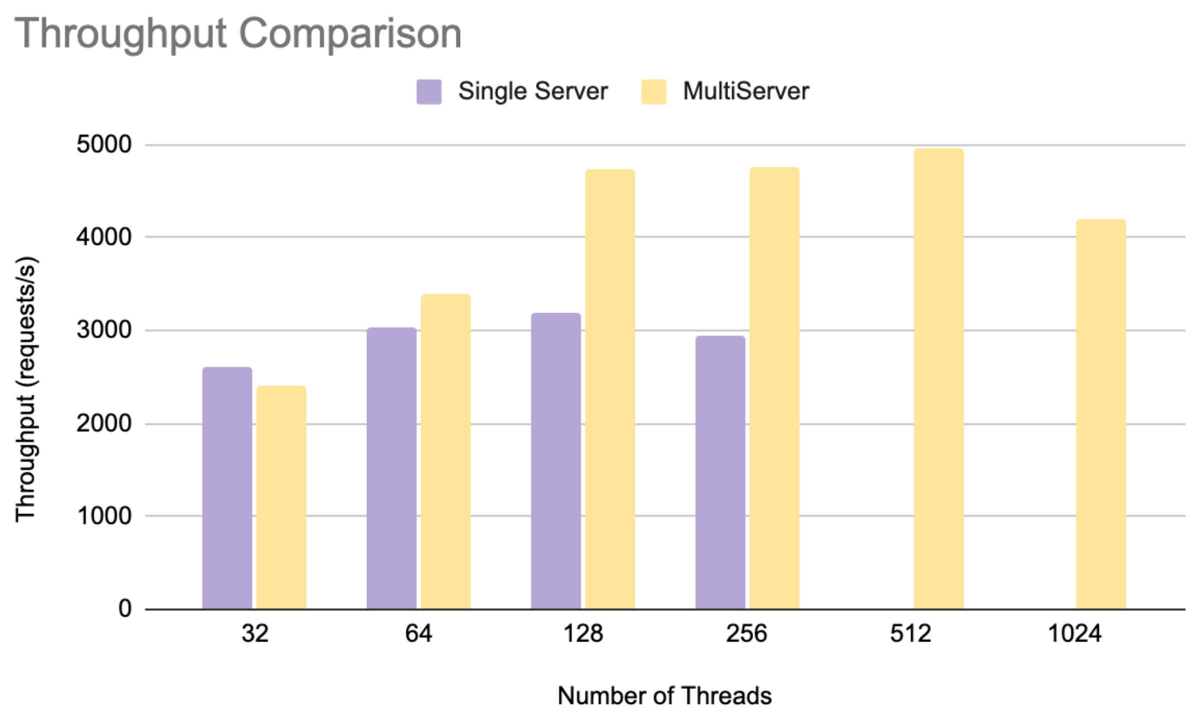
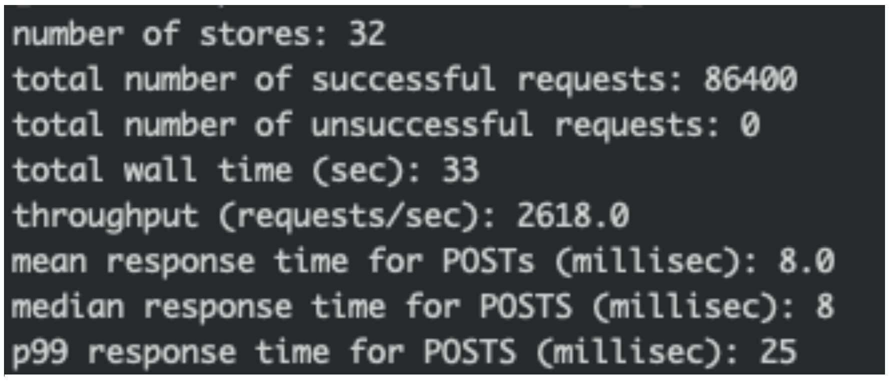


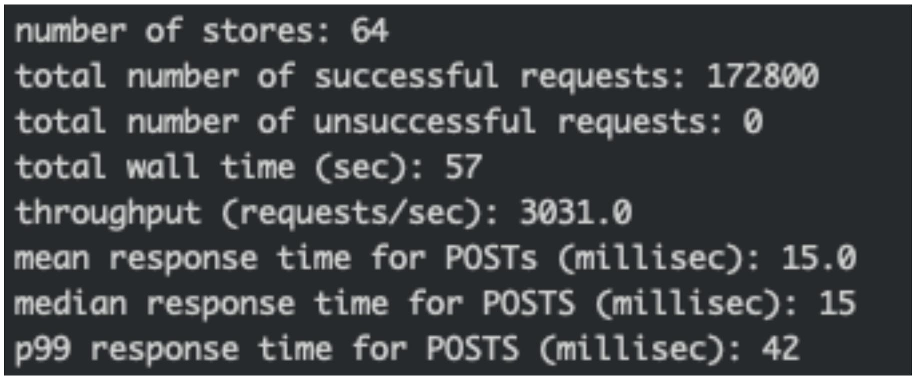
Figure 4. Single Server vs. Load Balanced Server Throughput Comparison

Appendix



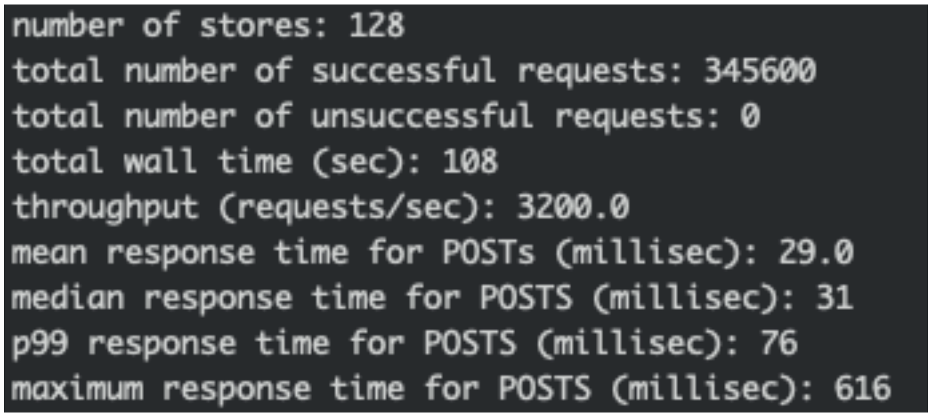
```
number of stores: 32
total number of successful requests: 86400
total number of unsuccessful requests: 0
total wall time (sec): 33
throughput (requests/sec): 2618.0
mean response time for POSTs (millisec): 8.0
median response time for POSTS (millisec): 8
p99 response time for POSTS (millisec): 25
```

Figure 5. Single Server Wall Time - 32 Stores



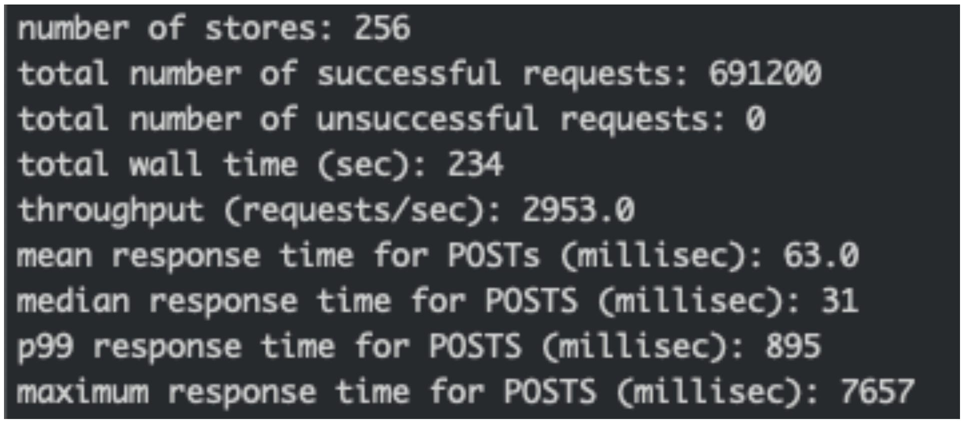
```
number of stores: 64
total number of successful requests: 172800
total number of unsuccessful requests: 0
total wall time (sec): 57
throughput (requests/sec): 3031.0
mean response time for POSTs (millisec): 15.0
median response time for POSTS (millisec): 15
p99 response time for POSTS (millisec): 42
```

Figure 6. Single Server Wall Time - 64 Stores



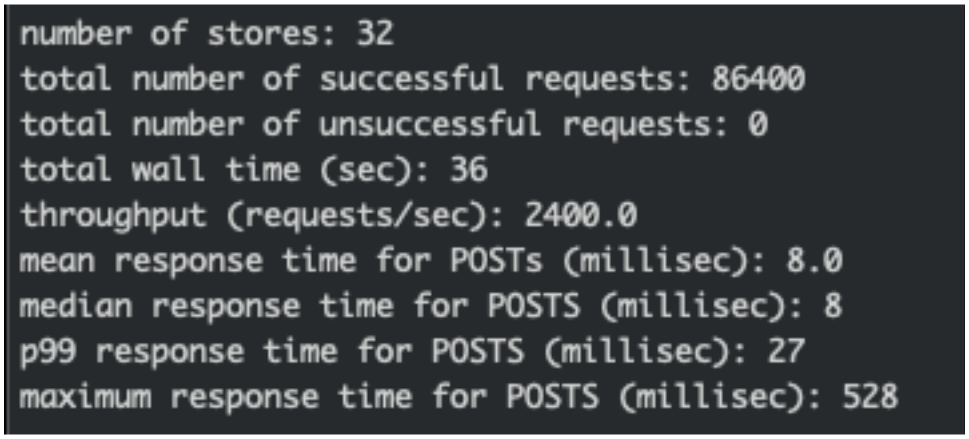
```
number of stores: 128
total number of successful requests: 345600
total number of unsuccessful requests: 0
total wall time (sec): 108
throughput (requests/sec): 3200.0
mean response time for POSTs (millisec): 29.0
median response time for POSTS (millisec): 31
p99 response time for POSTS (millisec): 76
maximum response time for POSTS (millisec): 616
```

Figure 7. Single Server Wall Time - 128 stores



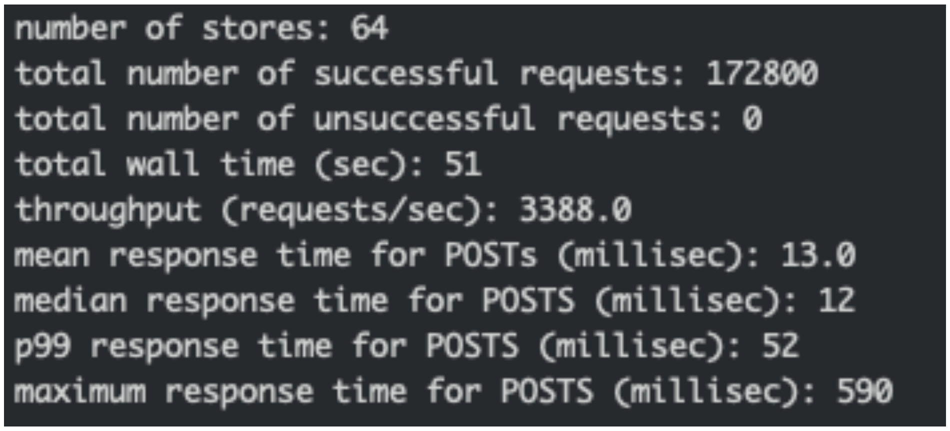
```
number of stores: 256
total number of successful requests: 691200
total number of unsuccessful requests: 0
total wall time (sec): 234
throughput (requests/sec): 2953.0
mean response time for POSTs (millisec): 63.0
median response time for POSTS (millisec): 31
p99 response time for POSTS (millisec): 895
maximum response time for POSTS (millisec): 7657
```

Figure 8. Single Server Wall Time - 256 stores



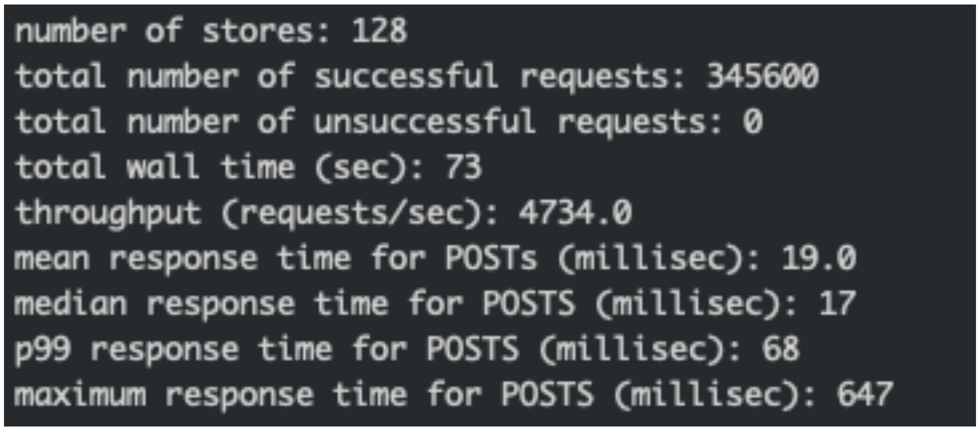
```
number of stores: 32
total number of successful requests: 86400
total number of unsuccessful requests: 0
total wall time (sec): 36
throughput (requests/sec): 2400.0
mean response time for POSTs (millisec): 8.0
median response time for POSTS (millisec): 8
p99 response time for POSTS (millisec): 27
maximum response time for POSTS (millisec): 528
```

Figure 9. Load Balanced Server Wall Time - 32 stores



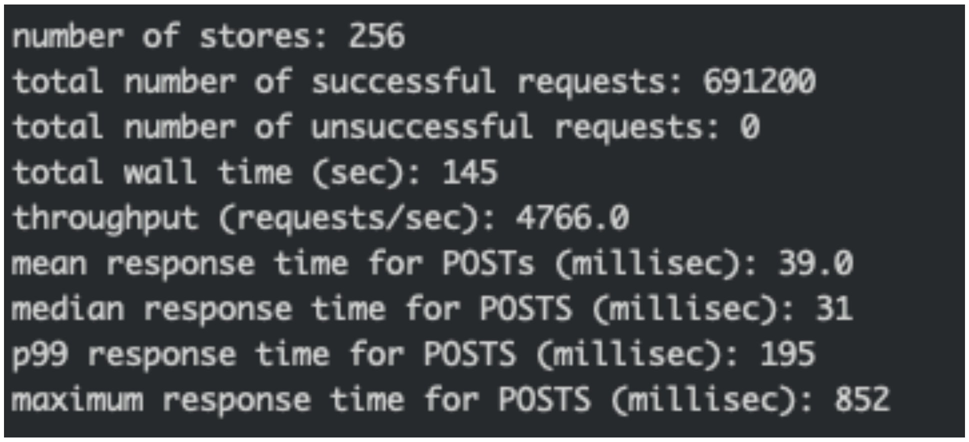
```
number of stores: 64
total number of successful requests: 172800
total number of unsuccessful requests: 0
total wall time (sec): 51
throughput (requests/sec): 3388.0
mean response time for POSTs (millisec): 13.0
median response time for POSTS (millisec): 12
p99 response time for POSTS (millisec): 52
maximum response time for POSTS (millisec): 590
```

Figure 10. Load Balanced Server Wall Time - 64 stores



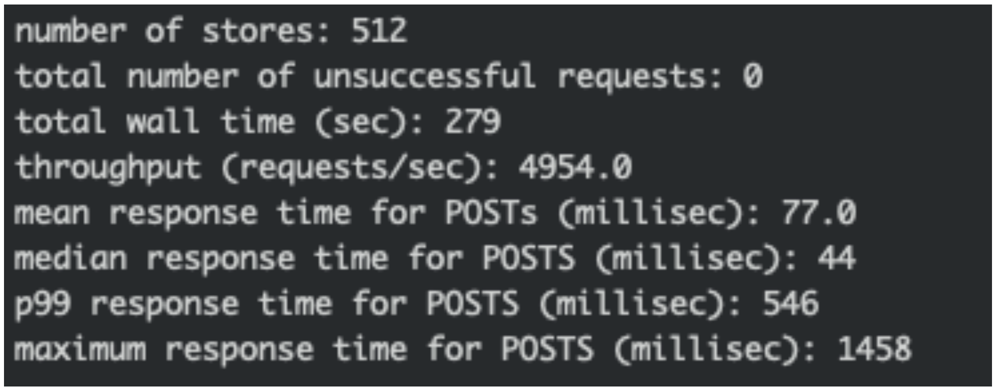
```
number of stores: 128
total number of successful requests: 345600
total number of unsuccessful requests: 0
total wall time (sec): 73
throughput (requests/sec): 4734.0
mean response time for POSTs (millisec): 19.0
median response time for POSTS (millisec): 17
p99 response time for POSTS (millisec): 68
maximum response time for POSTS (millisec): 647
```

Figure 11. Load Balanced Server Wall Time - 128 stores



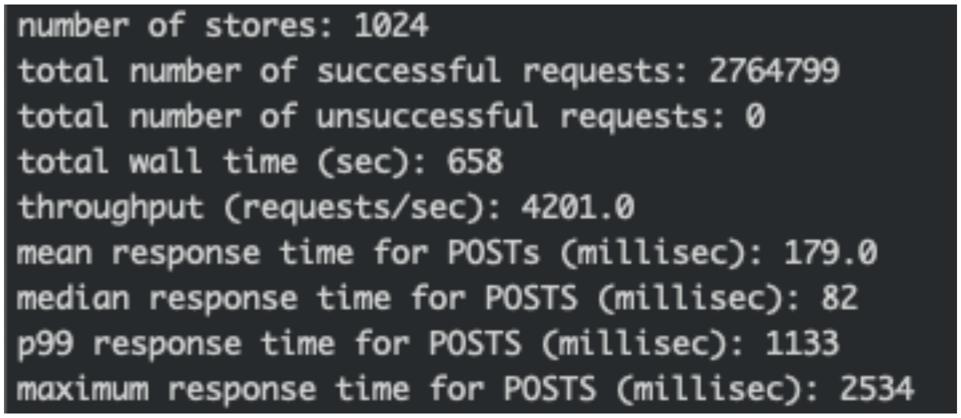
```
number of stores: 256
total number of successful requests: 691200
total number of unsuccessful requests: 0
total wall time (sec): 145
throughput (requests/sec): 4766.0
mean response time for POSTs (millisec): 39.0
median response time for POSTS (millisec): 31
p99 response time for POSTS (millisec): 195
maximum response time for POSTS (millisec): 852
```

Figure 12. Load Balanced Server Wall Time - 256 stores



```
number of stores: 512
total number of unsuccessful requests: 0
total wall time (sec): 279
throughput (requests/sec): 4954.0
mean response time for POSTs (millisec): 77.0
median response time for POSTS (millisec): 44
p99 response time for POSTS (millisec): 546
maximum response time for POSTS (millisec): 1458
```

Figure 13. Load Balanced Server Wall Time - 512 stores



```
number of stores: 1024
total number of successful requests: 2764799
total number of unsuccessful requests: 0
total wall time (sec): 658
throughput (requests/sec): 4201.0
mean response time for POSTs (millisec): 179.0
median response time for POSTS (millisec): 82
p99 response time for POSTS (millisec): 1133
maximum response time for POSTS (millisec): 2534
```

Figure 14. Load Balanced Server Wall Time - 512 stores