Building Scalable Distributed Systems

Assignment 4

Git Repository: https://github.com/akksshah/building-scalable-distributed-systems

Server Design Description

I rolled back the server code to what I had in assignment 2. With that I also replaced my database from a MySQL hosted on an RDS to DynamoDB. The biggest advantage that I had with this design was that I was able to achieve single digit millisecond latencies for almost all of my requests when I tried to save them directly at the server end. There is a static dynamoDbMapper object available to all the threads that are servicing the requests. It is thread safe and so all threads can use it to write the requests to the database.

Database design
I have a single table that has the key made out a combination of id as the primary hashkey and date as the sort key. I also two GSI (Global Secondary index): One with the index hashKey of customerId, and one with the index with storeId as the hashkey.

This database design makes it very easy to query for purchases made in a store or purchases made by a customer as we won't have to run a scan on the whole table. We can make effective use of partitions to reduce lookup times as we would be reading different partitions when querying for purchases.

Runs:

256 threads

```
[ec2-user@ip-172-31-58-56 ~]$ java -jar aakash/client.jar 256
================================================================================
Starting Execution at: 2021-04-25 14:08:08.411
Launching stores for eastern time
Launching stores for central time
Launching stores for pacific time
********** Statistics **********
Total number of stores used for simulation:
Total request sent: 691200
Total request successful: 691200
Total unsuccessful request: 0
Total wall time: 131s
Throughput: 5276.335877862595
Mean response time: 47.00767795138889ms
Median response time: 208.0ms
p99 (99 percentile): 290.0ms
Max response time: OptionalDouble[3448.0]ms
Ending execution at: 2021-04-25 14:10:20.890
================================================================================
```

Run with 512 threads

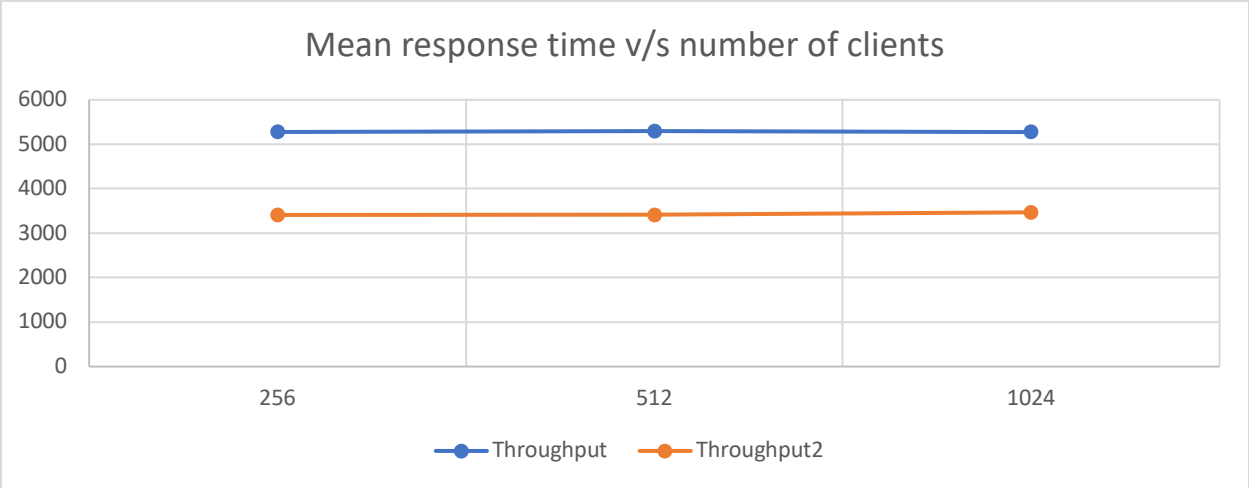
```
================================================================================
[ec2-user@ip-172-31-58-56 ~]$ java -jar aakash/client.jar 512
================================================================================
Starting Execution at: 2021-04-25 14:10:55.980
Launching stores for eastern time
Launching stores for central time
Launching stores for pacific time
********** Statistics **********
Total number of stores used for simulation:
Total request sent: 1382400
Total request successful: 1382400
Total unsuccessful request: 0
Total wall time: 261s
Throughput: 5296.551724137931
Mean response time: 92.33318070023148ms
Median response time: 92.5ms
p99 (99 percentile): 1283.0ms
Max response time: OptionalDouble[10019.0]ms
Ending execution at: 2021-04-25 14:15:19.366
================================================================================
[ec2-user@ip-172-31-58-56 ~]$
```

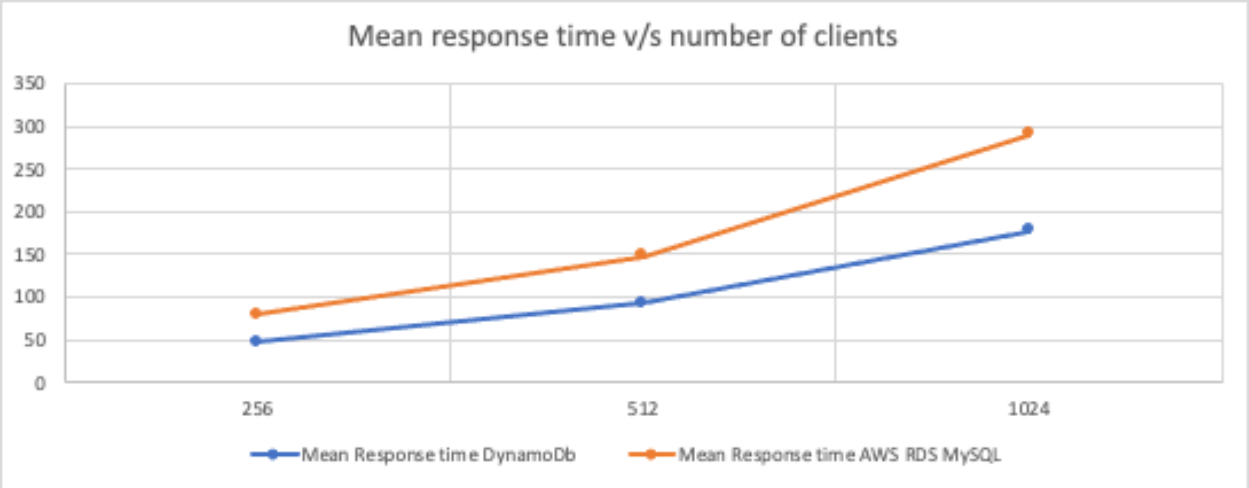
1024 threads

```
[ec2-user@ip-172-31-58-56 ~]$ java -jar aakash/client.jar 1024
================================================================================
Starting Execution at: 2021-04-25 13:12:09.253
Launching stores for eastern time
Launching stores for central time
Launching stores for pacific time
********** Statistics **********
Total number of stores used for simulation:
Total request sent: 2764800
Total request successful: 2764800
Total unsuccessful request: 0
Total wall time: 487s
Throughput: 5677.207392197125
Mean response time: 177.8702973090278ms
Median response time: 116.0ms
p99 (99 percentile): 939.0ms
Max response time: OptionalDouble[5474.0]ms
Ending execution at: 2021-04-25 13:20:20.403
================================================================================
```

Experiments and Inferences

Comparing the results that I had from assignment 2, what I essentially saw for the throughput for the client run for 256 threads is that the server simply provided a far better baseline throughput.





Simply switching out to a better database yield me more throughput. This was expected because Dynamo provides single digit millisecond latencies for all write request. For almost all of my request the time taken for a DynamoDB write was almost always less than 5ms.

Comparisons to Assignment 3.
My runs from assignment 3 showed the following results

Number of Threads	Throughput (RabbitMQ)	Mean Response Time (RabbitMQ)	Throughput (Dynamo)	Mean Response Time(Dynamo)
256	5619.51	45.086	5276.33	47.01
512	5421.17	92.84	5296.55	92.33

What we can see that the results from Assignment 3 mimic very close to that of Dynamo inserts in Assignment 4. The reason is that the cost to write the message to the RabbitMQ broker in Assignment 3 is almost very close enough to the write latencies provided by Dynamo.

So, what we can actually see is that by choosing the server design as Assignment 2, but replacing the database to dynamo, provides us with close enough throughput as having a RabbitMQ broker as in Assignment 3. The biggest advantage with the new system would be is that now this design would provide us with strong consistency of the database rather than that of Assignment 3 providing eventual consistency.

With the newer design, we can minimize the cost because in total we would be able to reduce the cost of having 2 separate ec2 instances that were otherwise running as in Assignment3. That is, we would not need an ec2 running the rabbit MQ broker and another to consume messages out of the broker. Also, we would have better fault tolerance because the RabbitMQ broker can become a single Point of failure if we don't have a backup broker which the servers can try in case of failures.

Depending on workload it is extremely easy for us to scale the system horizontally by replicating our servers in Assignment4 and registering them with the load balancer to ease out the load on each individual server. It is difficult to scale the Assignment 3 system, as the rabbit MQ broker can easily become a bottleneck. So, we would have to scale ourselves both the RabbitMQ broker and add individual servers at the same time and the consumers of the messages. Which in plain words simply would mean a far greater increase in cost than in Assignment4. It would be easy enough to say that scaling Assignment3 systems would incur a more significant increase in cost as compare to Assignment4.

Further testing (Scaling even further)

So, I tried to scale up even further by deploying an additional of 3 more servers and registering them with the load balancers and see the effect on throughput
I was able to achieve the following throughput
256 Threads

```
[ec2-user@ip-172-31-58-56 ~]$ java -jar aakash/client.jar 256
================================================================================
Starting Execution at: 2021-04-25 13:40:19.115
Launching stores for eastern time
Launching stores for central time
Launching stores for pacific time
********** Statistics **********
Total number of stores used for simulation:
Total request sent: 691200
Total request successful: 691200
Total unsuccessful request: 0
Total wall time: 97s
Throughput: 7125.773195876289
Mean response time: 34.813232060185186ms
Median response time: 9.0ms
p99 (99 percentile): 341.0ms
Max response time: OptionalDouble[3291.0]ms
Ending execution at: 2021-04-25 13:41:57.984
================================================================================
```

512 Threads

```
^[[D********** Statistics **********
Total number of stores used for simulation:
Total request sent: 1382400
Total request successful: 1382387
Total unsuccessful request: 13
Total wall time: 154s
Throughput: 8976.623376623376
Mean response time: 53.92656105324074ms
Median response time: 12.0ms
p99 (99 percentile): 1050.0ms
Max response time: OptionalDouble[10017.0]ms
Ending execution at: 2021-04-25 13:54:16.258
```

Factors	256 threads		512 threads	
	4 server	7 server	4 server	7 server
Throughput	5276.33	7125.77	5296.55	8976.62
Max Response Time	47.01ms	34.81ms	92.33ms	53.92ms
Median Response Time	208ms	9.8ms	92.5ms	12ms

Scaling up the servers for the same load, we can see that the max response time is quite impacted and that doubling the load from 256 to 512 does not have a huge impact. In fact, the max response time almost halves with the additional servers

The most impacted performance indicator is that the median time for servicing a request in both the loads, get served within single digit millisecond latencies which is an astonishing improvement compared to the load when they were served using 4 servers. We get an improvement of 95% percent under 256 thread load and about 90% improvement under the load of 512 threads.