CS6650 Assignment 3
Lena Duong
Fall 2022

Link to repo: https://github.com/lenad90/CS6650-Assignment3

Database Design

The database design I decided to implement is a simple key-value database using Redis. I stored the skierID as the key and the JSON string with the resort, day, season, vertical day, time, and liftID as the value using SADD. The SADD operation allows for collisions in the keys as long as the values are not duplicates. This made the most sense to me because the information of each individual skier is compartmentalized with unique values since realistically, data of a lift ride ticket would not have duplicates by the same individual. By doing so, the data model answers queries that require information about Skier N by locating the key with the SkierID. I chose this method as an alternative to storing each request with a single unique key (ex. An auto-increment as the key) because it offers a O(1) run time when querying Skier N, rather than searching O(n) through all the data values to find all of Skier N's information where n is the total number of requests. This was the most efficient solution for the queries that ask for information specific to Skier N. The last question of "how many unique skiers visited the resort X on day N" can be answered by querying all values where resort = X and day = N and count the amount of keys since each key is unique.

On the LiftServer side, I parsed the URL and extracted the information from the POST results into a new JSON string and sent it over to RabbitMQ. Storing the information as a JSON string was the best solution for me because the string can be easily deserialized to a JSONObject for the consumer to extract.

The Consumer initializes a JedisPool in order to allow multi-threaded connections to the Redis database. In the ConsumerRunnable, I would then receive the message from RabbitMQ, deserialize the message to retrieve the SkierID for the database's key, and then store the JSON string, created by the SkierDataModel object, as the value into the database. The JSON string is different from the JSON string that was sent by the RabbitMQ because it contains the VerticalDay and uses the SkierID as the key instead.

Deployment Topologies

I considered using ElastiCache to enable greater scalability, but I decided against it because the consumer's throughput perfectly matched the publisher's throughput when running them on the EC2 instances– the consumer was able to consume as many messages as the publisher was publishing, so a single node was sufficient. The Redis

database was installed onto the Consumer's instance, so the Consumer had to connect to its default host (127.0.0.1) at port 6379. To deploy, I would run the command "redis-server –daemonize yes –protected-mode no" on the Consumer's instance, the Consumer JAR file, and then my SkierClient. During the process, I would view RedisInsight to check if the data values are correctly inputted and the stabilizations of the throughputs on RabbitMQ.

The instance types were originally t2.micro, but after experimenting with different instance sizes, t3.medium gave the best throughput. I believe this is the case because t3 has four cores per instance as opposed to two and it offers greater storage capacity, which provides a faster throughput and lower latency overall. The more storage capacity was useful since the database itself consumed about 42 MB of memory and increase in network performance lowered the latency between the instances. As I experimented with different instance types, there was a gradual increase in throughput up until t3.medium. Any instance type afterwards did not show significant difference.

I decided to change all of the instances (LiftServer, Consumer, and RabbitMQ) to t3.medium because I noticed that the differing instance types (ex. T3.medium on consumer only) would affect the throughput and the consumer rate would not equal the publisher rate.

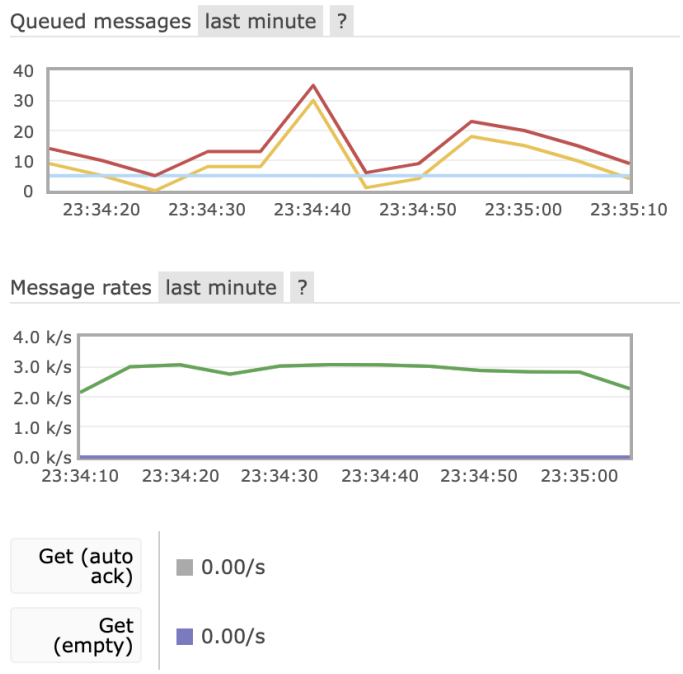
As seen from the screenshots below, the throughput before any increase in storage capacity was about 2600 requests per second. And after upgrading the instance type to t3.medium, the throughput almost doubled to 4000 requests per seconds, which was the highest I recorded.

Conclusion

After running multiple tests, I found that adjusting the consumer to 100 threads and increasing the storage capacity on all of the instances gave the best results. I decided not to implement the circuit breaker because my consumer and publisher throughputs were fairly stable, and the queues were short even before the mitigations. Implementing a circuit breaker didn't seem necessary since there were no spikes in requests that would negatively affect the performance, so slowing down the number of requests wouldn't help. On the other hand, increasing the storage capacity seemed like a more viable option since the t2.micro's network performance was low to moderate and had only one vCPU. The more vCPUs helped aid the multi threading process of the Consumer. The overall increase in processors, storage capacity, and network performance helped provide the best throughput for 200000 requests.

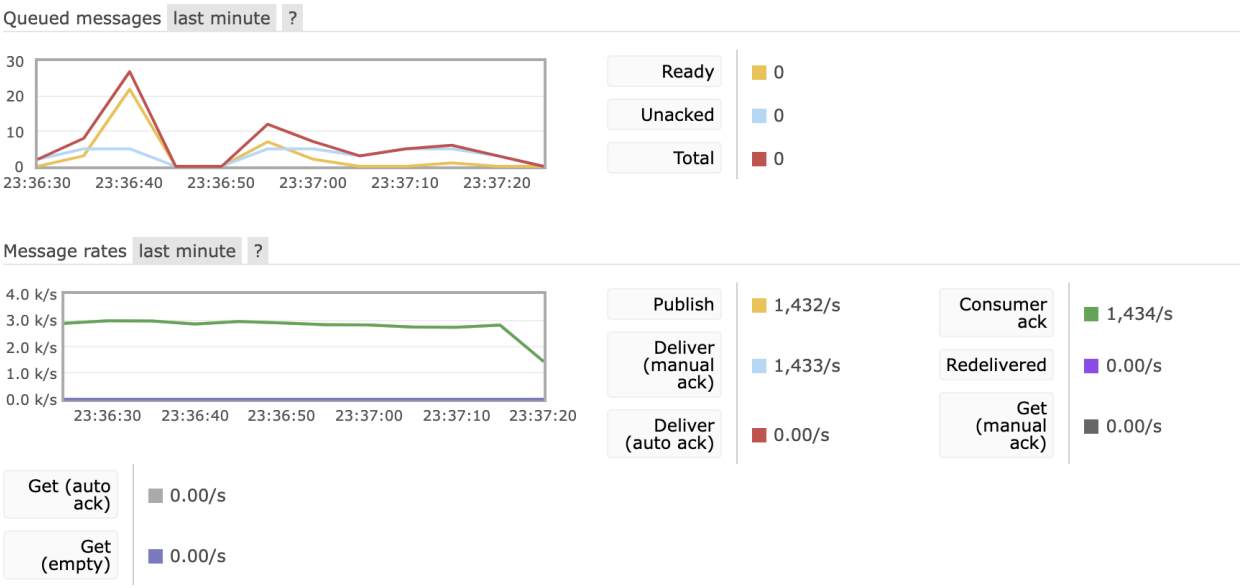
Before Mitigations

Skier Client 1




```
Number of successful POST requests: 51264
Number of unsuccessful POST requests: 0
Wall Time in seconds: 17.0
Actual Throughput = 51264/s for 64 threads
Expected throughput = 1447/s
==============Phase 3 Statistics ================
Number of successful POST requests: 64000
Number of unsuccessful POST requests: 0
Wall Time in seconds: 21.0
Actual Throughput = 64000/s for 128 threads
Expected throughput = 2895/s
==============Phase 4 Statistics ================
Number of successful POST requests: 52736
Number of unsuccessful POST requests: 0
Wall Time in seconds: 19.0
Actual Throughput = 52736/s for 256 threads
Expected throughput = 4523/s
============ PART 1 STATS =============
Total run time after all phases completed: 77/s
Total throughput after all phases completed: 2602/s
```

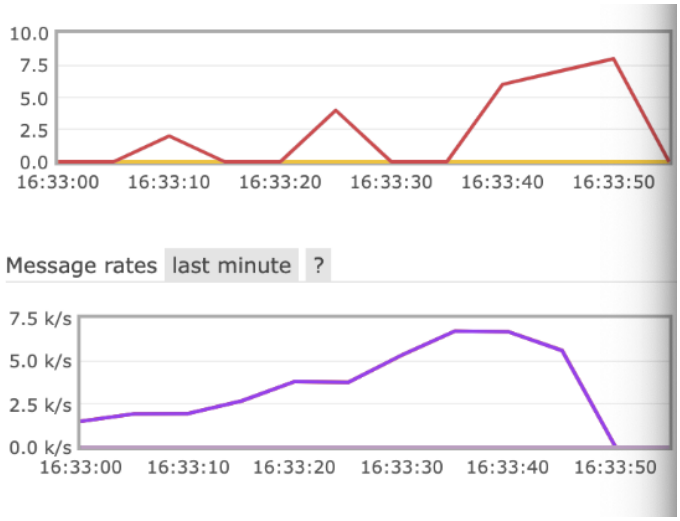
Skier Client 2




```
Phase 1 is starting ==========
Awaiting for thread to terminate
Phase 2 is starting ==========
Awaiting for thread to terminate
Phase 3 is starting ==========
Awaiting for thread to terminate
Awaiting for thread to terminate
Phase 4 is starting ==========
Awaiting for thread to terminate
============= PART 2 STATS ==============
Mean response time = 45.16398/ms
Median response time = 26.0/ms
Throughput = 2/ms
p99 Response Time = 440/ms
Min = 12.0/s Max = 3377.0/ms
```

After Mitigations

Skier Client 1




```
Number of successful POST requests: 51264
Number of unsuccessful POST requests: 0
Wall Time in seconds: 14.0
Actual Throughput = 51264/s for 64 threads
Expected throughput = 2812/s
==============Phase 3 Statistics ================
Number of successful POST requests: 64000
Number of unsuccessful POST requests: 0
Wall Time in seconds: 10.0
Actual Throughput = 64000/s for 128 threads
Expected throughput = 5623/s
==============Phase 4 Statistics ================
Number of successful POST requests: 52736
Number of unsuccessful POST requests: 0
Wall Time in seconds: 8.0
Actual Throughput = 52736/s for 256 threads
Expected throughput = 8786/s
============ PART 1 STATS =============
Total run time after all phases completed: 49/s
Total throughput after all phases completed: 4051/s
```

Skier Client 2


```
Phase 1 is starting ==========
Awaiting for thread to terminate
Phase 2 is starting ==========
Awaiting for thread to terminate
Phase 3 is starting ==========
Phase 4 is starting ==========
============= PART 2 STATS ==============
Mean response time = 22.139695/ms
Median response time = 18.0/ms
Throughput = 4/ms
p99 Response Time = 63/ms
Min = 11.0/s Max = 352.0/ms
```