Assignment 4 Writeup

Github:
https://github.com/AngelaJiang1225/CS6650/tree/master/A4

- **Load Testing (Test results for 256 and 512 clients)**

256 Threads, Single Server

With Read Replica(Assignment 4):

```
Number of successful requests= 386880
number of failed requests= 0
Total requests= 386880
Time Elapsed(ms)= 179014.0
Throughput(requests/sec)= 2161.1717519300164


Mean response time for GET= 182 ms
Median response time for GET= 73 ms
P99 response time for GET= 1244 ms
Max response time for GET= 1407 ms
Mean response time for POST= 239 ms
Median response time for POST= 195 ms
P99 response time for POST= 1588 ms
Max response time for POST= 2011 ms
```

Without Read Replica(Assignment 3)

```
Number of successful requests= 386880
number of failed requests= 0
Total requests= 386880
Time Elapsed(ms)= 224117.0
Throughput(requests/sec)= 1726.2412043709314


Mean response time for GET= 205 ms
Median response time for GET= 86 ms
P99 response time for GET= 1437 ms
Max response time for GET= 1509 ms
Mean response time for POST= 284 ms
Median response time for POST= 233 ms
P99 response time for POST= 1642 ms
Max response time for POST= 2177 ms
```

Without Read Replica(Assignment 2)

```
Number of successful requests= 386880
number of failed requests= 0
Total requests= 386880
Time Elapsed(ms)= 261108.0
Throughput(requests/sec)= 1481.6857392343397


Mean response time for GET= 236 ms
Median response time for GET= 102 ms
P99 response time for GET= 1677 ms
Max response time for GET= 1744 ms
Mean response time for POST= 311 ms
Median response time for POST= 285 ms
P99 response time for POST= 1920 ms
Max response time for POST= 2331 ms
```

256 Threads, Load Balanced

With Read Replica(Assignment 4):

```
Number of successful requests= 386880
number of failed requests= 0
Total requests= 386880
Time Elapsed(ms)= 154119.0
Throughput(requests/sec)= 2510.2680396317132


Mean response time for GET= 154 ms
Median response time for GET= 63 ms
P99 response time for GET= 1028 ms
Max response time for GET= 1277 ms
Mean response time for POST= 194 ms
Median response time for POST= 176 ms
P99 response time for POST= 1329 ms
Max response time for POST= 1851 ms
```

Without Read Replica(Assignment 3):

```
Number of successful requests= 386880
number of failed requests= 0
Total requests= 386880
Time Elapsed(ms)= 176119.0
Throughput(requests/sec)= 2196.6965517632966


Mean response time for GET= 179 ms
Median response time for GET= 71 ms
P99 response time for GET= 1238 ms
Max response time for GET= 1326 ms
Mean response time for POST= 219 ms
Median response time for POST= 183 ms
P99 response time for POST= 1256 ms
Max response time for POST= 1904 ms
```

Without Read Replica(Assignment 2):

```
Number of successful requests= 386880
number of failed requests= 0
Total requests= 386880
Time Elapsed(ms)= 218119.0
Throughput(requests/sec)= 1773.7106808668661


Mean response time for GET= 201 ms
Median response time for GET= 84 ms
P99 response time for GET= 1409 ms
Max response time for GET= 1533 ms
Mean response time for POST= 284 ms
Median response time for POST= 235 ms
P99 response time for POST= 1500 ms
Max response time for POST= 2191 ms
```

512 Threads, Single Server

With Read Replica(Assignment 4):

```
Number of successful requests= 773760
number of failed requests= 0
Total requests= 773760
Time Elapsed(ms)= 211785.0
Throughput(requests/sec)= 3653.516537998442


Mean response time for GET= 25 ms
Median response time for GET= 20 ms
P99 response time for GET= 186 ms
Max response time for GET= 854 ms
Mean response time for POST= 31 ms
Median response time for POST= 64 ms
P99 response time for POST= 973 ms
Max response time for POST= 3922 ms
```

Without Read replica(Assignment 3):

```
Number of successful requests= 773760
number of failed requests= 0
Total requests= 773760
Time Elapsed(ms)= 221707.0
Throughput(requests/sec)= 3490.011591875764


Mean response time for GET= 27 ms
Median response time for GET= 22 ms
P99 response time for GET= 201 ms
Max response time for GET= 943 ms
Mean response time for POST= 38 ms
Median response time for POST= 72 ms
P99 response time for POST= 1180 ms
Max response time for POST= 4258 ms
```

Without Read replica(Assignment 2):

```
Number of successful requests= 773760
number of failed requests= 0
Total requests= 773760
Time Elapsed(ms)= 249228.0
Throughput(requests/sec)= 3104.627088449131


Mean response time for GET= 41 ms
Median response time for GET= 32 ms
P99 response time for GET= 245 ms
Max response time for GET= 1108 ms
Mean response time for POST= 57 ms
Median response time for POST= 94 ms
P99 response time for POST= 1399 ms
Max response time for POST= 4872 ms
```

512 Threads, Load balanced

With Read replica(Assignment 4):

```
Number of successful requests= 773760
number of failed requests= 0
Total requests= 773760
Time Elapsed(ms)= 145764.0
Throughput(requests/sec)= 5308.306577755824


Mean response time for GET= 12 ms
Median response time for GET= 10 ms
P99 response time for GET= 103 ms
Max response time for GET= 498 ms
Mean response time for POST= 53 ms
Median response time for POST= 41 ms
P99 response time for POST= 855 ms
Max response time for POST= 2911 ms
```

Without Read replica(Assignment 3):

```
Number of successful requests= 773760
number of failed requests= 0
Total requests= 773760
Time Elapsed(ms)= 151688.0
Throughput(requests/sec)= 5100.996782870102


Mean response time for GET= 21 ms
Median response time for GET= 19 ms
P99 response time for GET= 204 ms
Max response time for GET= 843 ms
Mean response time for POST= 72 ms
Median response time for POST= 70 ms
P99 response time for POST= 1125 ms
Max response time for POST= 3197 ms
```

Without Read replica(Assignment 2):

```
Number of successful requests= 773760
number of failed requests= 0
Total requests= 773760
Time Elapsed(ms)= 184137.0
Throughput(requests/sec)= 4202.088662246045


Mean response time for GET= 34 ms
Median response time for GET= 28 ms
P99 response time for GET= 231 ms
Max response time for GET= 966 ms
Mean response time for POST= 85 ms
Median response time for POST= 79 ms
P99 response time for POST= 1208 ms
Max response time for POST= 3500 ms
```

- **Results Analysis**

Comparison of assignments 2, 3 with the results in assignment 4 for 256 and 512 threads:

Assignment 2:
Assignment 2 has the least throughput and costs more mean-median response time among the three assignments. A2 takes the strategy of increasing the number of database connections through the connection pool to make the throughput to be around 1400 requests/sec (256 threads) and 3000 requests/sec (512 threads).
Then, A2 scales out the application to use a load balanced server and deployed it on several instances. So there are large improvements in the response time and the application throughput (around 1700 requests/sec for 256 threads and 4000 requests/sec for 512 threads).

Assignment 3:
A3 used the strategy of RabbitMq as intermediate queue storage to resolve the problem of data inconsistency. RabbitMq introduces producer-consumer methods to write requests into the queue. Compared with not using RabbitMq, there is a significant increase in the throughput and response rate. Besides, cache is utilized through redis server to avoid duplicate requests received from the database time after time. The cache updates every time there are write requests for data.
By using RabbitMq to maintain eventual consistency and redis server for cache, the throughput reached over 1700 requests/sec(256 threads) and 3400 requests/sec(512 threads).
To further improve the performance of the application, A3 also applied a load balanced server and deployed it on multiple instances. Finally, the throughput reached over 2000 requests/sec(256 threads) and 5000 requests/sec(512 threads), which is a significant improvement.

Assignment 4:
A4 further utilized and implemented the READ Replica. The server was redesigned and restructured to route
the GET requests to the READ Replica, which always updates itself whenever writes requests are made to the primary database. The strategy reduced the pressure of heavy loading for database with GET requests parallel with the write requests.
Finally A4 achieved a throughput over 2000 requests/sec (256 threads with single server), 2500 requests/sec (256 threads with load balance), 3600 requests/sec (512 threads with single server) and 5100 requests/sec (512 threads with load balance), which is a huge progress compared with A2 and A3.

- **Database design**

a. Table `lifts_vertical`:

```
lift_id          int not null AUTO_INCREMENT primary key,
vertical         int  not null
```

b. Table `skier_records`:

```
id              int   not null AUTO_INCREMENT primary key,
resort_id       int                          not null,
lift_id         int                          not null,
season_id       int                          not null,
day_id          int                          comment 'day number in the
season',
day_time        int    default 0 not null,
CONSTRAINT fk_skier_records_lift_id
   FOREIGN KEY (lift_id)
       REFERENCES lifts_vertical(lift_id)
```

The database aims to improve the performance of reading and writing API's. The database implements some changes to make queries run faster and get higher throughput.

Read replica

This application utilizes Amazon RDS to create a special type of DB instance called "Read Replica" from a source DB instance in Mysql. This strategy is very effective to distribute the load of read and write requests when the server instance is overloaded. Besides, updates made to the primary DB instance are asynchronously copied to the read replica. Load on my primary DB instance can be reduced by routing read queries from my application to the read replica. Thus thanks to read replicas, I can elastically scale out beyond the capacity constraints of a single DB instance for overloaded databases.

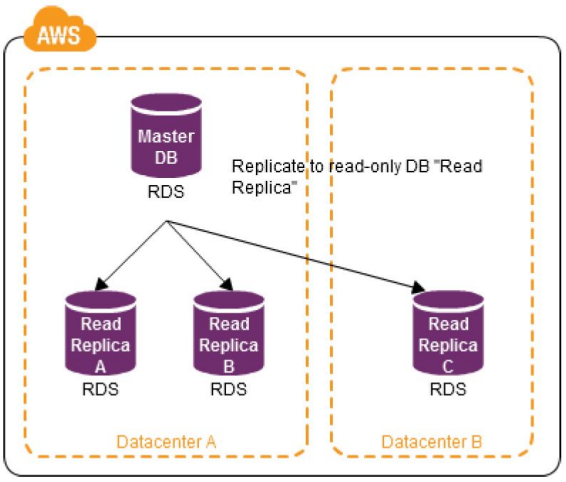
RDBMS service

The AWS Relational Database Management Systems (RDBMS) service has a function enabling the easy creation of a read-only database known as a "Read Replica." Here I created a read-only replica of the master database. When data is read by an application, access destination is set to the Read Replica.

AWS DMS and DynamoDB

AWS Data Migration Services (AWS DMS) is for migrating data from databases to DynamoDB tables. AWS DMS supports using Mysql as a source and AWS Schema Conversion Tool (AWS SCT) to convert my existing database schema from Mysql to DynamoDB.

Amazon DynamoDB is a key-value and document database with fully managed, multi-region, multi-active and durable database of built-in security. It has backup and in-memory caching for internet-scale applications. DynamoDB can handle more than 10 trillion requests per day and support peaks of more than 20 million requests per second. For this assignment, DynamoDB uses object mapping to transfer tables in Mysql into Amazon DynamoDB.



- **Application design**

1. **The Server Part**

API exposure
Skiers:
POST: /skiers/liftrides
GET: /skiers/{resortID}/days/{dayID}/skiers/{skierID}
GET: /skiers/{skierID}/vertical
Validation of input API
The data inside the url eg: {resortID}, {dayID}, {skierID} is validated to ensure that they are of proper types. The data input to the skiers post method is validated to make sure it is of type LiftRide. After validation passed, the data is processed using Dao layer and it returns application/json format.

Structure
dao layer:
This layer imports classes in entity and java-client-generated_2 packages plus ConnectionUtil class. ConnectionUtil class is used for establishing the connection and then operating implementing CRUD operations of skier_lift database such as creating SkierRecord, getting skier day vertical, getting skier total vertical and getting top ten verticals. The server uses RabbitMQ and subscribed consumers to keep consistency with the database. The RabbitMQ queue is used to temporarily persist the data and then many consumers are employed to pull data from the queue and persist it to the RDS, and I use MySQL database here.

connection layer:
This layer enables connection Pooling for RabbitMQ channels. A pool of channels is created in init() method so as to improve the input requests speed and response rate. This pool of channel is served from the pool.

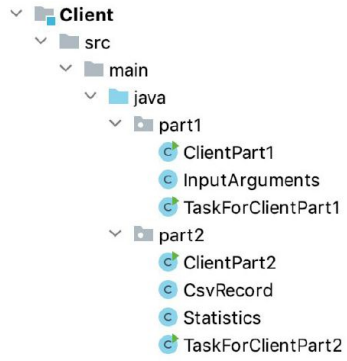
rabbitMQConnection layer:
This layer is for rabbitMQ to connect to JDBC. Then, it creates an instance of GenericObjectPool that enables our pool of connections. Besides, a ConnectionFactory object is created to be used by the pool to create the connection object. Finally a PoolableConnectionFactory object is created to wrap the Connection Object.

rabbitMQPubSub layer:

There are three classes used here: RabbitMQConsumerDao, SkiDataConsumer and SkiDataPublisher. This layer is used for writing lift rides to the database.
First for the SkiDataPublisher class, I set the number of messages per thread to 11 in it. The class publishes serialized liftRide objects to RabbitMQ queue, so that consumers can get and deserialize these lift rides in the queue.
SkiDataConsumer class imports SkierRecordsDao and implements the message queuing solution using RabbitMQ. The class creates SkierRecordsDao and uses liftRideDao.saveLiftRide method to save the lift ride object.

servlets layer:
This part is for accepting HTTP GET/POST request input and parsing url arguments. It imports classes in the dao layer. If parsed valid requests, the class will call methods with requested path and parameters, else If invalid, return response code such as 400/404.

2. **The Client Part**



In the client part: Post requests for each thread is 1000. Phase 3 threads have an extra 10 GETs for resort total in addition to 10 GETs for day totals. Dao layer is imported in client part, classes in swagger package such as ApiClient, ApiResponse are used in TaskForClientPart to write liftRide with http information.