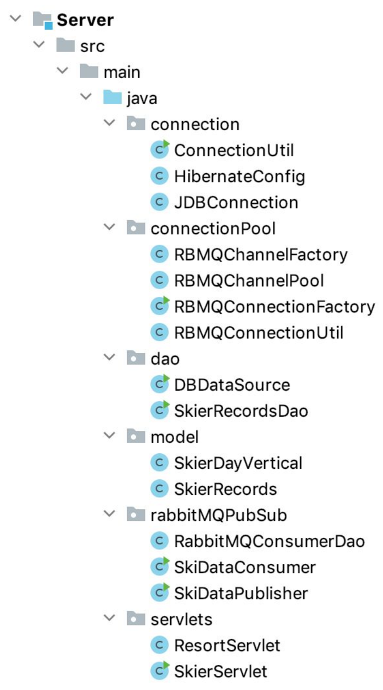
Assignment 3 Report

Github:
https://github.com/AngelaJiang1225/CS6650

1. **Application design:**

- **The Server Part Diagram**

Server design:



```
Server
  src
    main
      java
        connection
          ConnectionUtil
          HibernateConfig
          JDBConnection
        connectionPool
          RBMQChannelFactory
          RBMQChannelPool
          RBMQConnectionFactory
          RBMQConnectionUtil
        dao
          DBDataSource
          SkierRecordsDao
        model
          SkierDayVertical
          SkierRecords
        rabbitMQPubSub
          RabbitMQConsumerDao
          SkiDataConsumer
          SkiDataPublisher
        servlets
          ResortServlet
          SkierServlet
```

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.

- **The 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)
```

JDBC Connection and pooling

My program supports buth dbcp and hibernate to get connection to database. It allows pooling features and ensures the max number of connections to the database is 15 at a time.

The application uses getSessionFactory() method to implement session. Then, dao layer utilizes the session to build connections for each query.

- **The Server Design**

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:

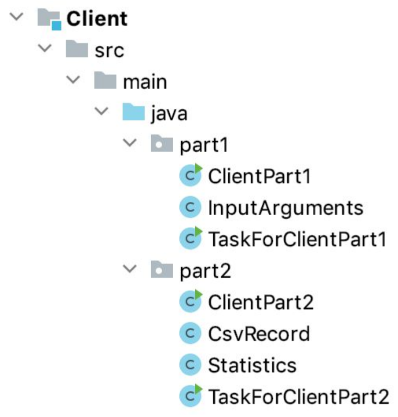
There are three classes used here: RabbitMQConsumerDao, SkiDataConsumer and SkiDataPublisher. This layer is used for writing lift rides to the database.

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.

- **The Client Part**

Client design

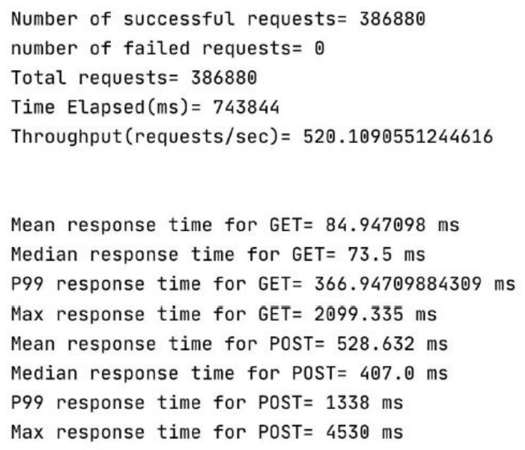


```
Client
  src
    main
      java
        part1
          ClientPart1
          InputArguments
          TaskForClientPart1
        part2
          ClientPart2
          CsvRecord
          Statistics
          TaskForClientPart2
```

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.

- **Results comparison**

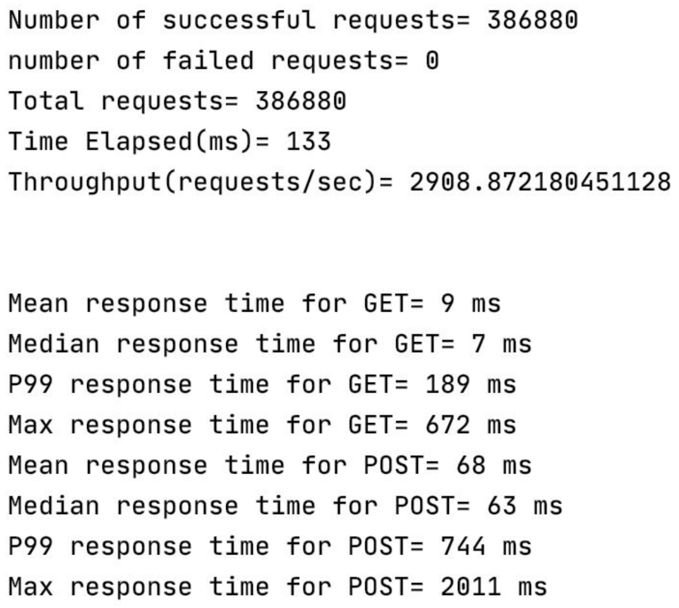
(1) Thread 256(single server)
Assignment 2:



```
Number of successful requests= 386880
number of failed requests= 0
Total requests= 386880
Time Elapsed(ms)= 743844
Throughput(requests/sec)= 520.1090551244616


Mean response time for GET= 84.947098 ms
Median response time for GET= 73.5 ms
P99 response time for GET= 366.94709884309 ms
Max response time for GET= 2099.335 ms
Mean response time for POST= 528.632 ms
Median response time for POST= 407.0 ms
P99 response time for POST= 1338 ms
Max response time for POST= 4530 ms
```

Assignment 3 (Non-persistent Message Queue):

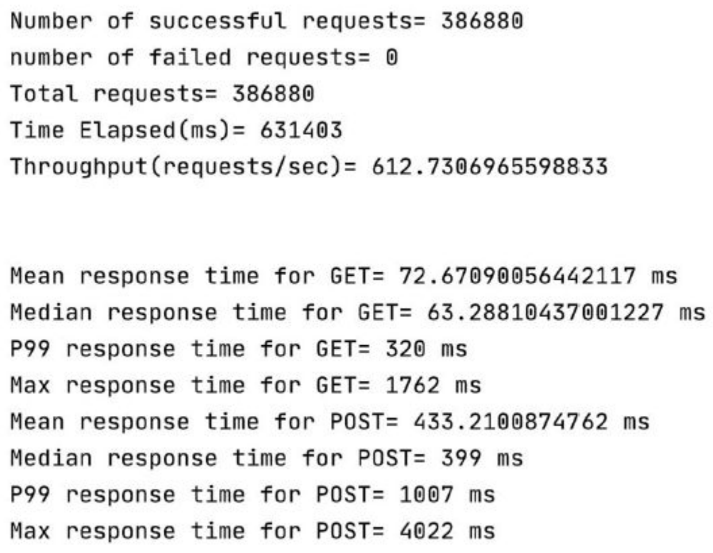


```
Number of successful requests= 386880
number of failed requests= 0
Total requests= 386880
Time Elapsed(ms)= 133
Throughput(requests/sec)= 2908.872180451128


Mean response time for GET= 9 ms
Median response time for GET= 7 ms
P99 response time for GET= 189 ms
Max response time for GET= 672 ms
Mean response time for POST= 68 ms
Median response time for POST= 63 ms
P99 response time for POST= 744 ms
Max response time for POST= 2011 ms
```

(2) Thread 256(load balanced)

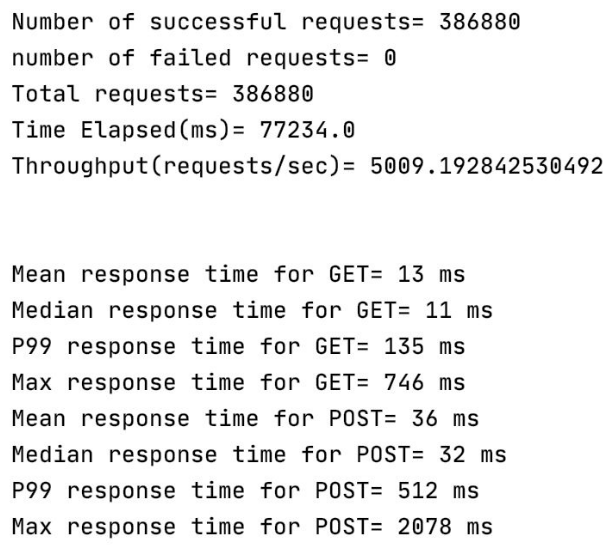
Assignment 2:



```
Number of successful requests= 386880
number of failed requests= 0
Total requests= 386880
Time Elapsed(ms)= 631403
Throughput(requests/sec)= 612.7306965598833


Mean response time for GET= 72.67090056442117 ms
Median response time for GET= 63.28810437001227 ms
P99 response time for GET= 320 ms
Max response time for GET= 1762 ms
Mean response time for POST= 433.2100874762 ms
Median response time for POST= 399 ms
P99 response time for POST= 1007 ms
Max response time for POST= 4022 ms
```

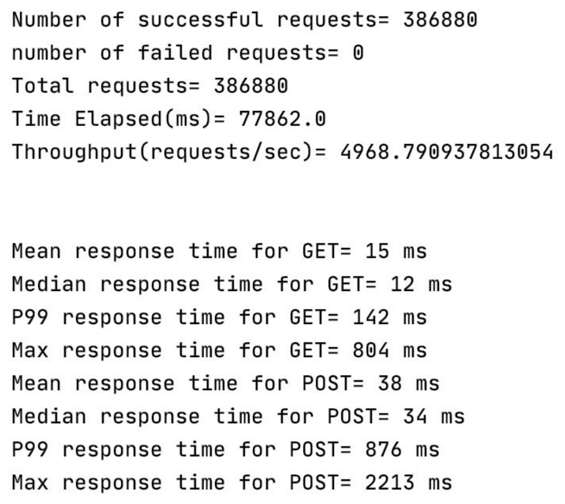
Assignment 3 (Non-persistent Message Queue)



```
Number of successful requests= 386880
number of failed requests= 0
Total requests= 386880
Time Elapsed(ms)= 77234.0
Throughput(requests/sec)= 5009.192842530492


Mean response time for GET= 13 ms
Median response time for GET= 11 ms
P99 response time for GET= 135 ms
Max response time for GET= 746 ms
Mean response time for POST= 36 ms
Median response time for POST= 32 ms
P99 response time for POST= 512 ms
Max response time for POST= 2078 ms
```

Assignment 3 (persistent Message Queue)

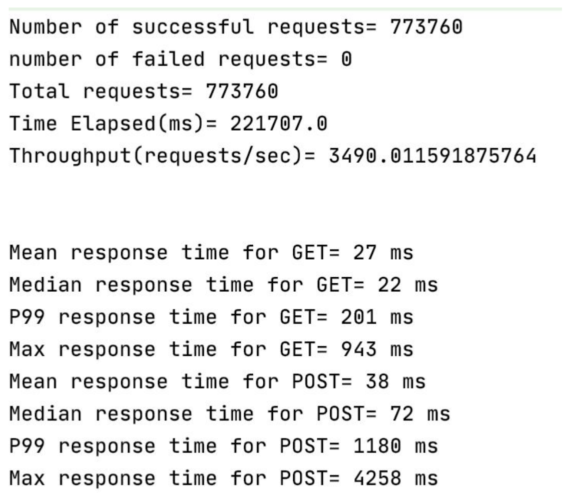


```
Number of successful requests= 386880
number of failed requests= 0
Total requests= 386880
Time Elapsed(ms)= 77862.0
Throughput(requests/sec)= 4968.790937813054


Mean response time for GET= 15 ms
Median response time for GET= 12 ms
P99 response time for GET= 142 ms
Max response time for GET= 804 ms
Mean response time for POST= 38 ms
Median response time for POST= 34 ms
P99 response time for POST= 876 ms
Max response time for POST= 2213 ms
```

(3) 512 threads

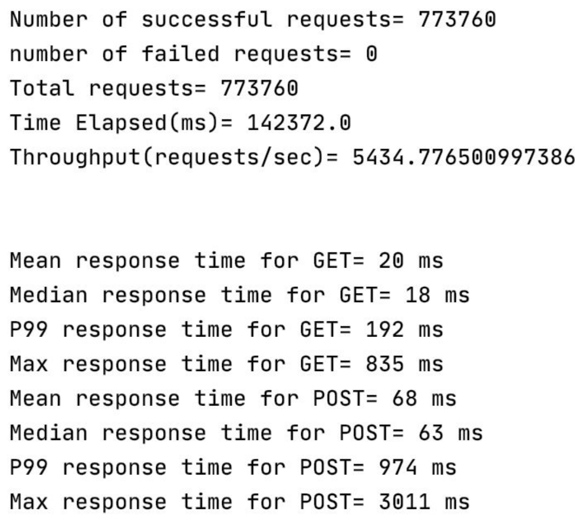
single server(non-persistent)



```
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
```

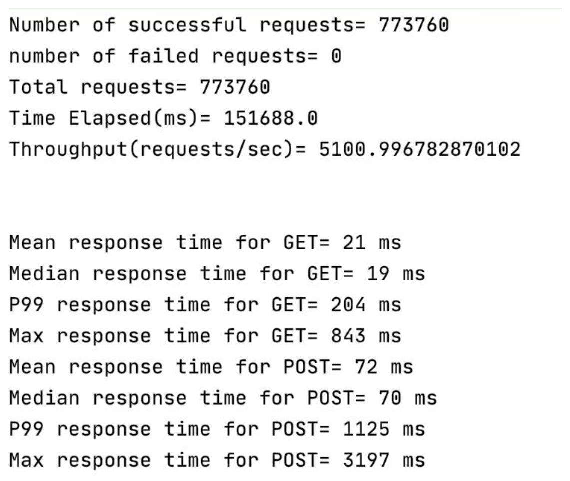
Load balanced(non-persistent)



```
Number of successful requests= 773760
number of failed requests= 0
Total requests= 773760
Time Elapsed(ms)= 142372.0
Throughput(requests/sec)= 5434.776500997386


Mean response time for GET= 20 ms
Median response time for GET= 18 ms
P99 response time for GET= 192 ms
Max response time for GET= 835 ms
Mean response time for POST= 68 ms
Median response time for POST= 63 ms
P99 response time for POST= 974 ms
Max response time for POST= 3011 ms
```

Load balanced(persistent)



```
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
```

2. Discussion

- **Number of Consumer Thread and Instances**

The consumer related classes speed up the performance when a GET request is sent. Firstly it checks if the skierID already exists, “yes” means that the skierID is cached and there is no need to access the database, we only need to update the vertical for that.
I adjusted the number of consumers many times and found that 6 consumers fit my application best. I also tested different numbers of threads and found that 11 is the biggest number I tested to best the performance. So finally I set 6 consumers and implemented 11 threads to run each of them with 8 pooled DB connections.

- **Comparison with Assignment 2 (256 Thread)**

From the results shown above, we can see that both throughput and response time in 256 thread have great improvement in Assignment 3 using RabbitMQ's message broker capabilities compared with Assignment 2. The single server utilizing the message queue (non-persistent) had a throughput that was more than 5 times of the throughput seen with a single server without the messaging queue from assignment 2(from 520 requests/sec to 2908 requests/sec). Besides, the POST response time for the message queue assisted server is approximately 6 times faster than the server without the message queue(from 407ms down to 63ms).
This goes similar with the load balanced tests, although not that massive. With load balancing, the server throughput was increased by a factor of 8, while the POST response rates decreased by a factor over 10.

- **Comparison of 512 thread and 256 thread(with message queue)**

Single Server

From the results above, we can see that single server in 256 threads performs worse than that of 512 threads, which surprised me. After further study and consideration, I understand that that may be because 512 thread amount of requests has higher order than that of 256 thread. Without load balancing, the single server may fail to undertake requests and spend more time dealing with SocketTimeoutExceptions and client request retries that damage the performance.

Load Balanced and persistent/non-persistent

It shows that the load balancer increases the throughput, mean and median response rates of the application.I also find that load balance increases the throughput in both assignment 2 and assignment 3. In terms of results of persistent and non-persistent queues, the improvement in throughputs using the persistent queue is very small compared with non-persistent queues.