A. GitHub: https://github.com/jiehuizhou66/cs6650

B. Server Design:

- Overall Design:
  In this assignment, the major component change in the server is to add a producer_consumer based queueing solution between the servelet and database. I choose to use RabbitMQ for my implementation.

- New added / updated classes:

1. Sender.java

   This is the producer class which produce and publish the message into the queue. It has two methods:

   a. Sender constructor(): it make connection to the RabbitMQ, create a channel, and declare a queue. The queue can be persistent or non-persistent queue depending on the "durable" boolean when declaring the queue.
   b. publishMsg(): it receives the message from the SkierServelet and publish the message to the queue. The message can also been set to persistent or non-persistent message. Set both queue and message to persistence is to ensure the message won't get lost if queue is down. All the message in the queue can be resumed when the queue gets restarted.

2. Receiver.java

   This is the consumer class which responsible for receiving message from queue and then process the message. In my implementation, I make connnection to the queue in the main method. And then run multiple threads in order to increase the capability of message(or request) processing.

3. ReceiverThread.java

   This is the consumer thread class which process the message that consumer class received. It has following methods:

   a. ReceiverThread constructor(): A channel to the RabbitMQ is created and the queue is declared. Same as what Sender class does in its constructor, the queue is set to be persistent or non-persistent by using the boolean "durable". To let the queue be either persistent or non-persistent, it needs to be set in both producer and consumer end. I set allowance of the number of message that RabbitMQ can send to the client at one time to be 100.
   b. Run(): The deliver callback is been declared in this method. The deliver callback will be invoked by the RabbitMQ and receive message from the queue. In the deliver callback, it calls the createLiftRide() to write message to

the database. If write failed, it will publish the message back to the queue to ensure the message won't be lost. Once the write to DB transaction completed, I send an acknowledgement to the queue so queue can remove this message.

c. CreateLiftRide(): It extract the data from the message(Json object), encapsulate the data to a LiftRide instance, and invoke the createLiftRide in LiftRideDao class.

4. SkierServlet.java:

   It is the main class which receive and respond to the request from Skiers API. The major changes in this assignment are:

   a. Init() method is added to initiate the producer(in my code is Sender). So the connection to the RabbitMQ is kept open.
   b. Instead of call LiftRideDao to write dat to the database directly, in the doPost() it let producer(in my code is Sender) to publish the request to the RabbitMQ.

- <u>Existing classes:</u>

5. LiftRide.java

   It is a POJO class which stores ResortID, DayID, SkierID, Time, LiftID.

6. LiftRideDao.java

   The main functionalities of this class is to write/update/read from database.

   a. createLiftRide() write the LiftRide entry in the database. It write one row in database every time this method gets invoked. I tried to optimize the method to to batch transaction for optimization. However, it performs worse than single write transaction(the time takes longer). So I revert back to write one single row approach.
   b. queryLiftRideVerticalPerDay() finds the corresponded LiftRide data from the database based on skierID and dayID.
   c. queryLiftRideVerticalPerResort() finds the corresponded LiftRide data from the database based on skierID and resortID.

7. DBCPDataSource.java

It is the connection manager class between my program and MySQL database. I increased the initial pool size and the maxTotal in order to optimize the performance of database operations.

8. Schema:

```
CREATE TABLE LiftRides(
   id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
        resortId VARCHAR(255) NOT NULL,
        dayId VARCHAR(255) NOT NULL,
   skierId VARCHAR(255) NOT NULL,
   time VARCHAR(255) NOT NULL,
   liftId VARCHAR(255) NOT NULL,
   index resortIndex (skierId,resortId),
        index dayIndex (skierId,dayId)
);
```

C. Result Comparison:

In the assignment 3, the server implemented the producer consumer based solution using RabbitMQ. Let's see compare the data from A2 to check out if the performance is improved by using this approach.

1. Single Server A3 vs A2:

1.1. 256 threads screenshot (from A3):

```
-----------------Part 1------------------
number of successful requests sent: 386880
number of unsuccessful requests sent: 0
Wall time: 528290
Throughput: 732
-----------------Part 2------------------
mean response time (millisecs): 357.7729365100197
median response time (millisecs): 33.81
throughput (total number of requests/wall time): 732.325049
p99 response time (99th percentile, millisecs): 201230.18925
max response time(millisecs): 224850.153
-----------------SkierDayVerticalGET------------------
mean response time (millisecs): 69.1487603221008
median response time (millisecs): 38.07911451
p99 response time (99th percentile, millisecs): 8627.94264237
-----------------SkierResortVerticalGET------------------
mean response time (millisecs): 124.97504721829586
median response time (millisecs): 23.294622532
p99 response time (99th percentile, millisecs): 1679.6017354
```

1.2. A3 server (RabbitMQ Persistent queue) vs A2 server:

1.2.1. Table

single server	assignement #	wall time (s)	throughput	median response (ms)
	A2 256 threads	1341.607	288	106
	A3 256 threads	528.29	732	33.81

1.2.2. Observations:

As we can see the throughput is improved significantly in A3 for test run with 256 maximum threads. My understanding of this performance improvement is because the response time for POST request from server to client is being reduced by using producer_consumer based solution. In A2, the server will send response back to the client when the request being write to the database successfully or failed to write into the database. So time of process the request takes a big fragment of the response time.

For the implementation in A3, the SkierServlet will send the response code back to client once the producer finish publishing the request to the RabbitMQ. In the consumer end, if the database transaction failed, it will publish the failed request back to the queue to ensure the request is being write to the database. The response time from server to client is being reduced in this way.

2. Load Balancer A3 vs A2:

2.1. 256 threads screenshot (from A3):

```
----------------Part 1-----------------
number of successful requests sent: 386880
number of unsuccessful requests sent: 0
Wall time: 477808
Throughput: 809
----------------Part 2-----------------
mean response time (millisecs): 287.7099875930521
median response time (millisecs): 27.0
throughput (total number of requests/wall time): 809.6976191273483
p99 response time (99th percentile, millisecs): 183937.0
max response time(millisecs): 183948.0
----------------SkierDayVerticalGET-----------------
mean response time (millisecs): 57.52946428571428
median response time (millisecs): 24.0
p99 response time (99th percentile, millisecs): 7872.0
----------------SkierResortVerticalGET-----------------
mean response time (millisecs): 92.3515625
median response time (millisecs): 17.0
p99 response time (99th percentile, millisecs): 1225.0
```

2.2. A3 server (RabbitMQ Persistent queue) vs A2 server

2.2.1. Table

load balancer	assignement #	wall time (s)	throughput	median response (ms)
	A2 256 threads	661.085	585	83
	A3 256 threads	477.808	809	27

2.2.2. Observations:

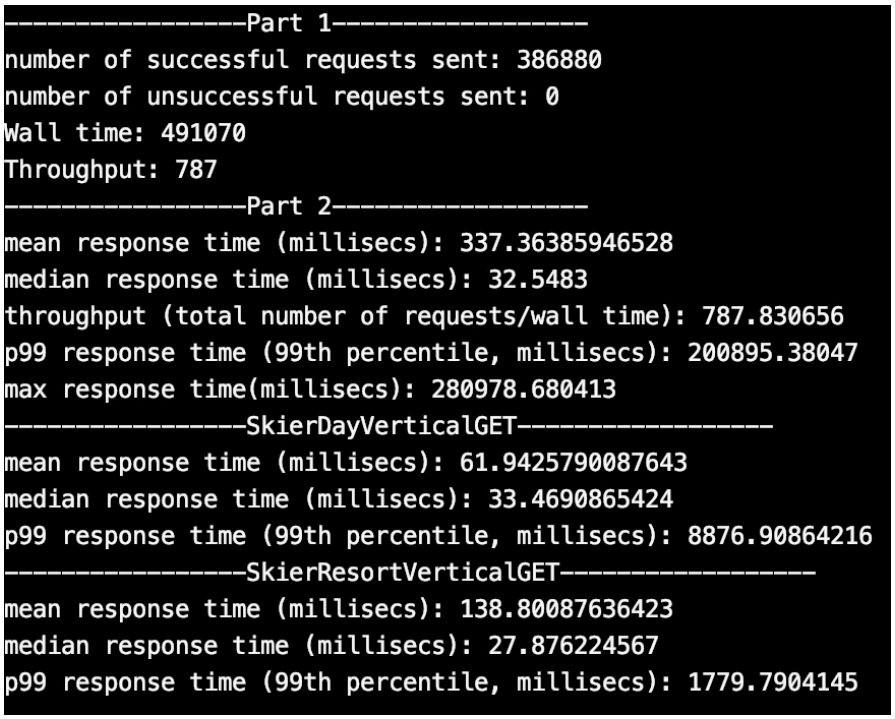
As we can see the performance of load balance test in A3 is better than the load balance test in A2 for maximum 265 threads test run.

Compare load balance test to single server test in A3, my data shows that the performance is just slightly better in load balance test. Based on my result, I assume that doing load balance is not very necessary as we see the performance of server end is not effectively improved.

For the thread of consumer which response for writing data to the database, I tried to run 2, 5, 8, 12 number of threads. I did not see a big difference between 8 threads and 12 threads. So I keep to run 8 threads of consumer for each consumer instance.

3. Persistent Queue vs. Non-persistent Queue

3.1 Screenshot of Non-persistent queue (the screenshot of persistent queue can be found in 1.1):



3.2 Table

assignement #	wall time (s)	throughput	median response (ms)
persistent queue 256 threads	528.29	732	33.81
non-persistent queue 256 threads	491.07	787	32.54

3.3 Observations:

The throughput of using non-persistent queue is slightly better than using persistent queue. However since the difference is not significant and the data point is limited, I cannot conclude that using non-persistent queue will provide better throughput.

4. 256 threads vs.512 threads

4.1 Screenshot of 512 threads single server with persistent queue implementation. The screenshot of 256 threads can be found in 1.1.:

```
------------------Part 1------------------
number of successful requests sent: 773760
number of unsuccessful requests sent: 0
Wall time: 897753
Throughput: 861
------------------Part 2------------------
mean response time (millisecs): 401.58974235799
median response time (millisecs): 94.7995
throughput (total number of requests/wall time): 861.885173
p99 response time (99th percentile, millisecs): 206790.246
max response time(millisecs): 238045.246805
------------------SkierDayVerticalGET-------------------
mean response time (millisecs): 116.36700766
median response time (millisecs): 89.5673401
p99 response time (99th percentile, millisecs): 9946.3580086
------------------SkierResortVerticalGET-------------------
mean response time (millisecs): 98.205756383
median response time (millisecs): 72.27496543
p99 response time (99th percentile, millisecs): 4953.58096
```

4.2 Table:

thread #	wall time (s)	throughput	median response (ms)
256 threads	528.29	732	33.81
512 threads	897.753	861	94.79

4.3 Observations: As we can see in the result, the throughput of 512 threads test with persistent queue is slightly increased compare to 216 threads test with persistent queue.