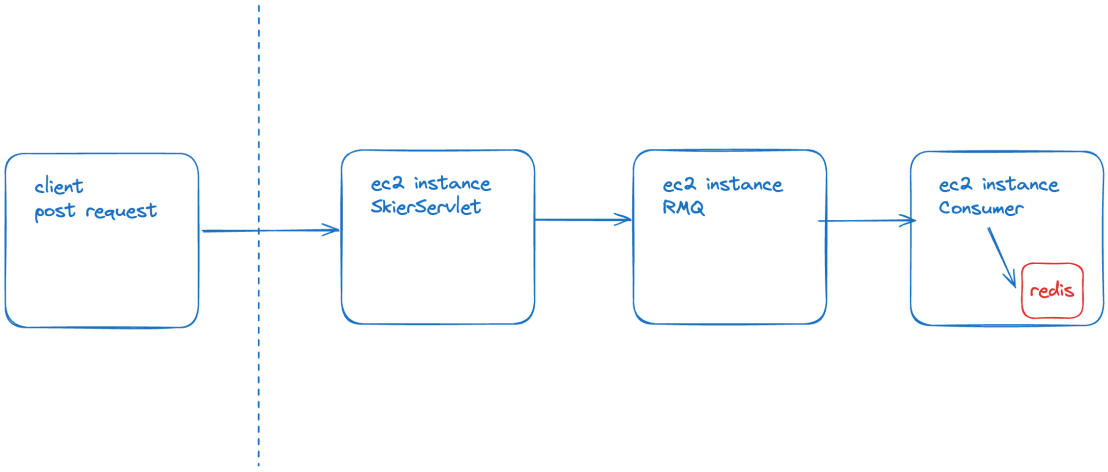
Repo: https://github.com/wakaka1123/A3DAO

A3 report

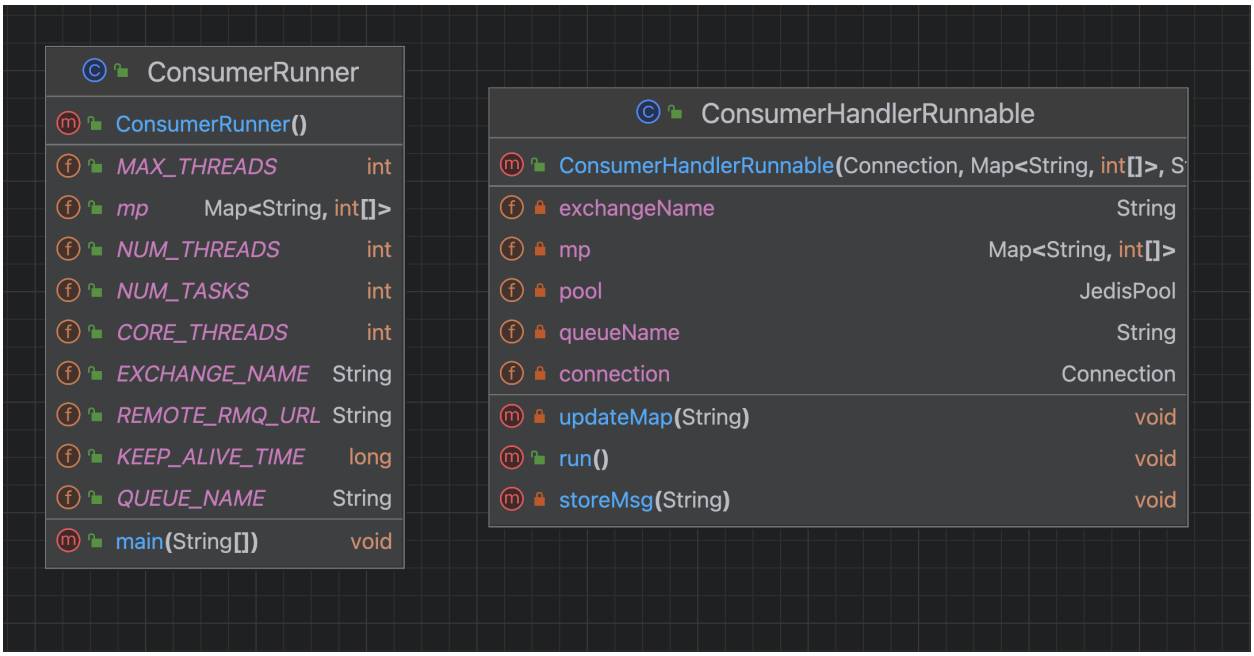
Step1 Add a database

For this assignment, I chose Redis as persistent data storage to ensure performance of writing to the database. Being a key-value store, Redis is more widely used as cache for query. To fit our assignment of persistent storage, I designed 4 data model in response to the 4 queries as stated in the assignment description specifically.



In the above architecture, load balancer fronting the SkierServlet is removed for budget limitation and in the following step, we will see throttling is implemented instead of increasing capacity in ec2 instance for this cause as well. Redis is set up as local service on the consumer on port 6379. We could set up multiple ec2 consumers writing to Redis server running on a separate instance, but that might again quickly drain our budget for the data transmission. We will keep the architecture as above for this lab.

Class layout



ConsumerRunner: main thread to run consumers in a customized thread pool.

ConsumerHandlerRunnable: consumer thread to handle the message-consuming task and store data.

Data model design

Recall that we have the following fields available when taking message out of RMQ.

[time, liftID, resortID, seasonID, dayID, skierID]

Eg [217, 10, 1, 2024, 1, 123]

We will use Redis data types to save the data in the format of 4 types of key-value pairs and provide corresponding queries.

My original design is to store skierID as key and the message in JSON as value. It will be good in terms of compacting Q1 to Q3 to one code block when writing to Redis, but in reverse it will requires iterating JSON objects in searching for an answer to one of the queries whose overhead will be costly when more data is stored.

- Q1: For skier N, how many days have they skied this season?

Redis data structure:
Set
key: "sk" + skierID + "se" + seasonID + "dSet"
value: dayID
Query: SCARD <key>

- Q2: For skier N, what are the vertical totals for each ski day? (calculate vertical as liftID*10)

Redis data structure:
Hash
key: "sk" + skierID + "se" + seasonID + "vMap"
field: "d" + dayID
value: vertical
Query: HGET <key> <field>

- Q3: For skier N, show me the lifts they rode on each ski day?

Redis data structure:
Set
key: "sk" + skierID + "se" + seasonID + "d" + dayID + "lSet"
value: liftID
Query: SMEMBERS <key>

- Q4: How many unique skiers visited resort X on day N?

Redis data structure:
Set
key: "res" + resortID + "d" + dayID + "skSet"
value: skierID
Query: SCARD <key>

Step2 Run tests

Test1 Configurations:

Client: max of 200 threads sending 200k reqs in total

Consumer: 200 consumers waiting in RMQ

```
----------Part 1 starts----------
199817 requests are successful
183 requests are failed
Number of threads initially: 32, max number of threads: 200
Wall time: 169481 ms
Throughput: 1180 reqs/s
----------Part 2 starts----------
Mean response time: 22 ms
Median response time: 19 ms
Throughput: 1183 reqs/s
99th percentile response time: 87.0 ms
Min response time: 10 ms
Max response time: 667 ms

Process finished with exit code 0
```

Test1 Analysis:

We can see some failed request amounting 0.92% of total reqs and queue size amounting to 13+. I tried to implement circuit breaker on the server side to mitigate the issue.

The breaker's config is to check every 5 seconds if the request handling is as high as 7500 and it will throttle until it comes down to 6000 every 5 seconds. With this setting, the range of reqs/s should be lower than 1500 reqs/s.

```
EventCountCircuitBreaker(7500, 5, TimeUnit.SECONDS, 6000);
```

Test2 Configurations:
Client: max of 200 threads sending 200k reqs in total
Consumer: 200 consumers waiting in RMQ
Server side circuit breaker: 7500 reqs max in 5 sec

```
----------Part 1 starts----------
199835 requests are successful
165 requests are failed
Number of threads initially: 32, max number of threads: 200
Wall time: 175383 ms
Throughput: 1140 reqs/s
----------Part 2 starts----------
Mean response time: 24 ms
Median response time: 18 ms
Throughput: 1142 reqs/s
99th percentile response time: 89.0 ms
Min response time: 11 ms
Max response time: 5405 ms

Process finished with exit code 0
```

Test2 Analysis:
Throttling is working in terms of controlling throughputs, but we can still find out there are 160+ failed reqs and queue size around 10+.

It could be client side racing to send reqs but there is not enough consumers handling the reqs. Then I modified the breaker on the servlet as follows allowing a higher throughput.

```
EventCountCircuitBreaker(10000, 5, TimeUnit.SECONDS, 7500);
```

And I reduced the number of threads sending reqs to 100 on the client side while remain the number of total reqs as 200k, hoping that more consumers handling reqs than clients sending reqs on every second can help to improve the consuming capacity and thus mitigate the failed reqs.

Test3 Configurations:

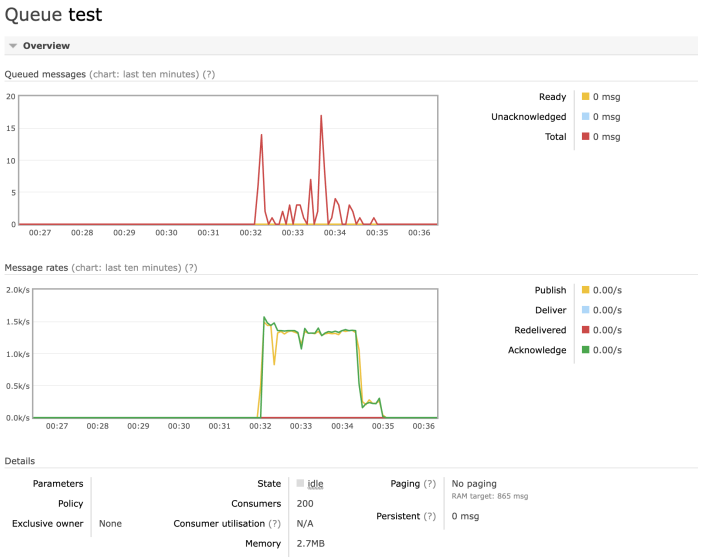
Client: max of 100 threads sending 200k reqs in total

Consumer: 200 consumers waiting in RMQ

Server side circuit breaker: 10000 reqs max in 5 sec

```
----------Part 1 starts----------
199795 requests are successful
205 requests are failed
Number of threads initially: 32, max number of threads: 100
Wall time: 182247 ms
Throughput: 1097 reqs/s
----------Part 2 starts----------
Mean response time: 22 ms
Median response time: 19 ms
Throughput: 1098 reqs/s
99th percentile response time: 78.0 ms
Min response time: 11 ms
Max response time: 5207 ms

Process finished with exit code 0
```



Test3 Analysis:

The throughput remains unaffected of the increased threshold of circuit breaker as expected. And the failed reqs and queue size problem pertains.

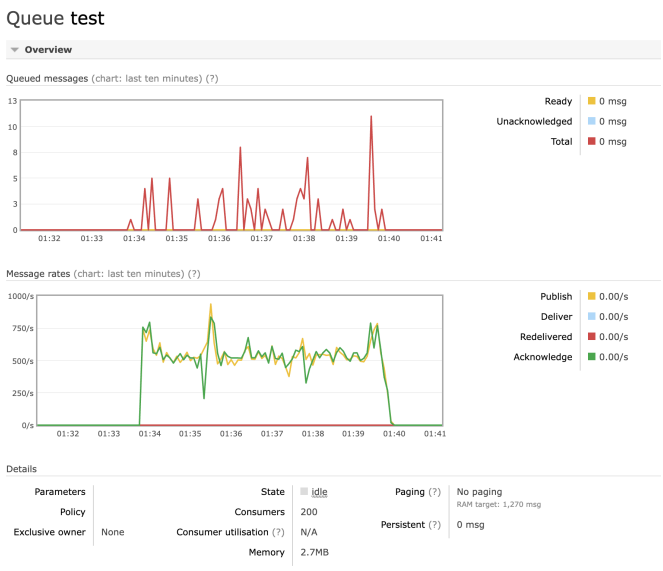
In response to that, circuit breaker is implemented on the client side when retry time reaches 4 and will be blocked for 1 sec before it reached the 5th retry and throws exception.

```
EventCountCircuitBreaker(4, 1,
    TimeUnit.SECONDS, 4, 1, TimeUnit.SECONDS);
```

Test4 Configurations:
Client: max of 100 threads sending 200k reqs in total
Consumer: 200 consumers waiting in RMQ
Server side circuit breaker: 10000 reqs max in 5 sec
Client side circuit breaker: 4 retries hold for 1 sec

```
----------Part 1 starts----------
199964 requests are successful
36 requests are failed
Number of threads initially: 32, max number of threads: 100
Wall time: 368259 ms
Throughput: 543 reqs/s
----------Part 2 starts----------
Mean response time: 54 ms
Median response time: 16 ms
Throughput: 543 reqs/s
99th percentile response time: 1612.0 ms
Min response time: -54 ms
Max response time: 10240 ms

Process finished with exit code 0
```



Test4 Analysis:
With circuit breaker on client side, the failed reqs declined to 30+ and queue size under 10. But it also adds limit to the throughput to 500+ reqs/s and greatly extend request handling time.

In response to this matter, I try to recover the number of client threads to 200 to find out if it will improve the throughput.

Test5 Configurations:

Client: max of 200 threads sending 200k reqs in total

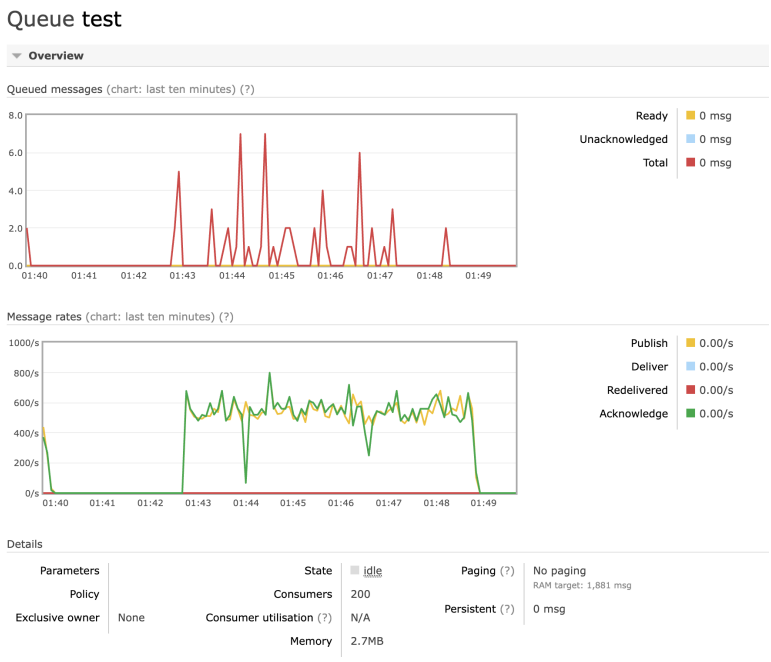
Consumer: 200 consumers waiting in RMQ

Server side circuit breaker: 10000 reqs max in 5 sec

Client side circuit breaker: 4 retries hold for 1 sec

```
----------Part 1 starts----------
199957 requests are successful
43 requests are failed
Number of threads initially: 32, max number of threads: 200
Wall time: 371010 ms
Throughput: 539 reqs/s
----------Part 2 starts----------
Mean response time: 56 ms
Median response time: 16 ms
Throughput: 540 reqs/s
99th percentile response time: 1888.0 ms
Min response time: 10 ms
Max response time: 10219 ms

Process finished with exit code 0
```



Test5 Analysis:

Failed reqs number is 40+ while queue size under 10. Throughputs is not improving.

Test Summary

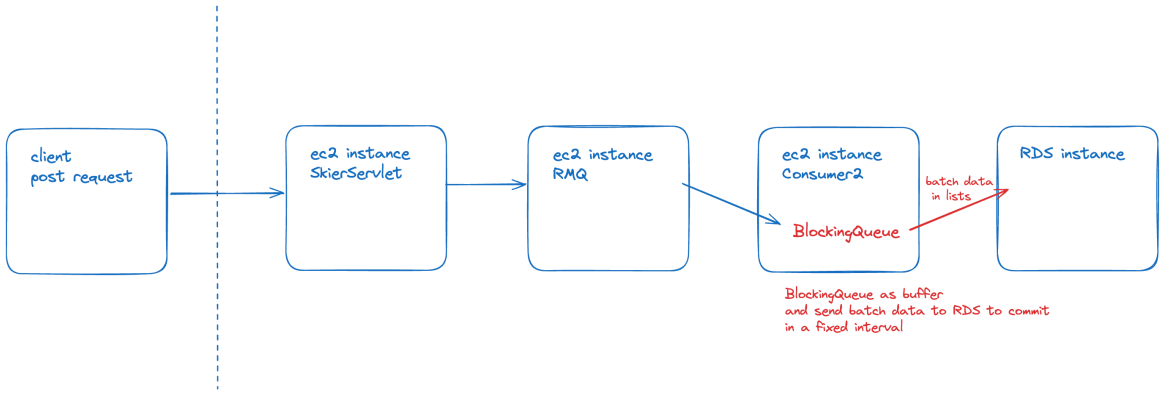
	Test Config					Metrics			
No.	Client threads	Total reqs	consumers	Client breaker	Servlet breaker	Throughputs	Wall time	Failed reqs	Queue size
Test1	200	200k	200	NA	NA	1180 reqs/s	169s	183	10~15
Test2	200	200k	200	NA	1500reqs/s	1140 reqs/s	175s	165	5~13
Test3	100	200k	200	NA	2000reqs/s	1097 reqs/s	182s	205	5~16
Test4	100	200k	200	4 retries 1 sec	2000reqs/s	543 reqs/s	368s	36	5~10
Test5	200	200k	200	4 retries 1 sec	2000reqs/s	540 reqs/s	371s	43	3~7

In other words, client side breaker has helped to mitigate the number of failed requests. Providing more consumer threads versus client thread, a wide range server side breaker (i.e. 2000 reqs/s max) does slight effect on failed request number and queue size.

I also tried server side breaker with setting like 1000 reqs/s max, but that will only hinder the throughput but not help to reduce the failed reqs.

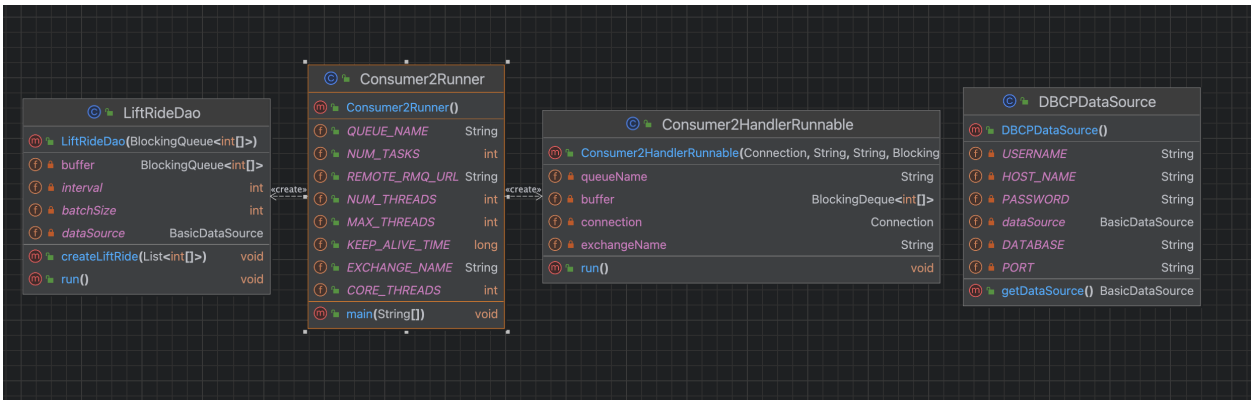
Step1 Add a database: Remake

In order to enable more flexible query on skier data set not excluding to the 4 questions mentioned in assignment requirement. I implemented a relational database as persistent storage using RDS MySQL.



Reference is from: https://github.com/Peihao-Zhu/cs6650/blob/main/Assignment3/conusmer2/src/main/java/LiftRideDao.java

Class layout



LiftRideDao: single thread task to fetch batch data from buffer and send local batch to MySQL to insert into LiftRides table in the LiftRides database.

Consumer2Runner: class to start resources for batch write

Consumer2HandleRunnable: class to get fetch data from RMQ and add to the buffer

DBCPDataSource: util class to get connection to MySQL

Data model design

Instance: skier's entry

Columns: [id, skierID, resortID, seasonID, dayID, time, liftID]

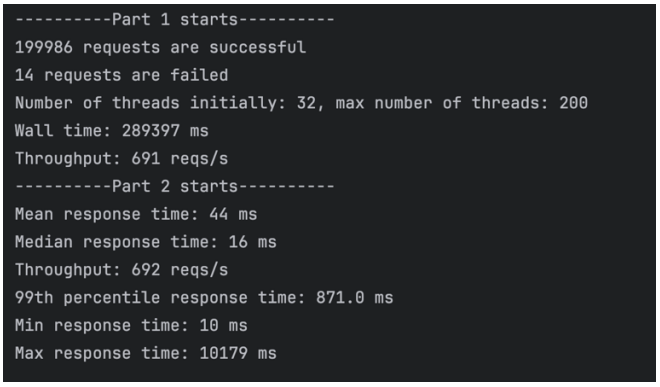
```
MySQL [LiftRides]> describe LiftRides;
+----------+------+------+-----+---------+----------------+
| Field    | Type | Null | Key | Default | Extra          |
+----------+------+------+-----+---------+----------------+
| id       | int  | NO   | PRI | NULL    | auto_increment |
| skierId  | int  | YES  |     | NULL    |                |
| resortId | int  | YES  |     | NULL    |                |
| seasonId | int  | YES  |     | NULL    |                |
| dayId    | int  | YES  |     | NULL    |                |
| time     | int  | YES  |     | NULL    |                |
| liftId   | int  | YES  |     | NULL    |                |
+----------+------+------+-----+---------+----------------+
```

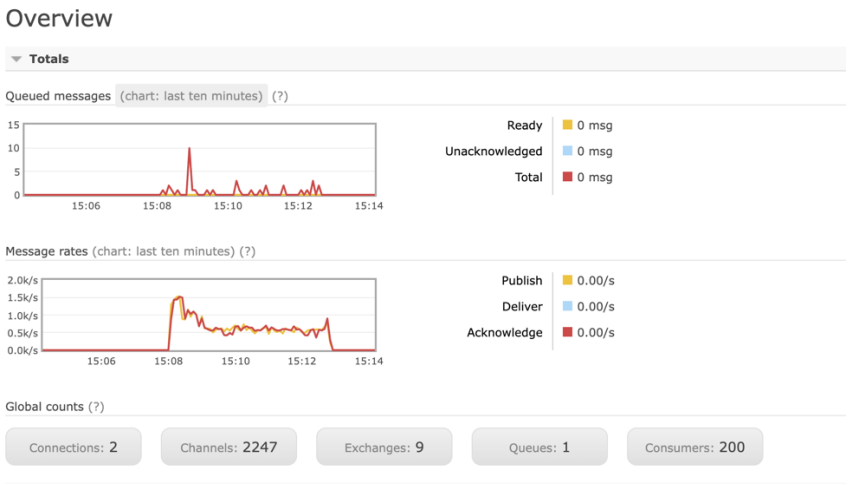
Reason for setting up a unique id field is that random generated skier's data entry can have duplicates in other fields. But we still want to keep all the data including the duplicate ones in RDS MySQL.

Q1~4 Query SQL: omitted here, not main purpose.

Step2 Run tests

Test6 Configurations same as Test5:
Client: max of 200 threads sending 200k reqs in total
Consumer: 200 consumers waiting in RMQ
Server side circuit breaker: 10000 reqs max in 5 sec
Client side circuit breaker: 4 retries hold for 1 sec






```
Database changed
MySQL [LiftRides]> select count(*) from LiftRides;
+----------+
| count(*) |
+----------+
|   199986 |
+----------+
1 row in set (0.03 sec)
```

Test6 Analysis:

The count(*) tells that all successful data request is stored in database.

Throughputs and failed requests better than Test5 but it could be due to network. What's worth mentioning is that due to the timed batch write to MySQL, it will take longer for the data to be transferred to the database. This can be improved to start a thread pool and make multiple batch write tasks to submit to the pool.

Renewed Test Summary

	Test Config					Metrics			
No.	Client threads	Total reqs	consumers	Client breaker	Servlet breaker	Throughputs	Wall time	Failed reqs	Queue size
Test1(redis)	200	200k	200	NA	NA	1180 reqs/s	169s	183	10~15
Test2(redis)	200	200k	200	NA	1500reqs/s	1140 reqs/s	175s	165	5~13
Test3(redis)	100	200k	200	NA	2000reqs/s	1097 reqs/s	182s	205	5~16
Test4(redis)	100	200k	200	4 retries 1 sec	2000reqs/s	543 reqs/s	368s	36	5~10
Test5(redis)	200	200k	200	4 retries 1 sec	2000reqs/s	540 reqs/s	371s	43	3~7
Test6(MySQL)	200	200k	200	4 retries 1 sec	2000reqs/s	691 reqs/s	289s	14	3~10