

Assignment 3 - Yingbing Zhu

NUID: 001402648

Github: <https://github.com/Yingbing-Zhu/cs-6650-distributed-assignment-3/>

Database Choice: MySQL using AWS RDS

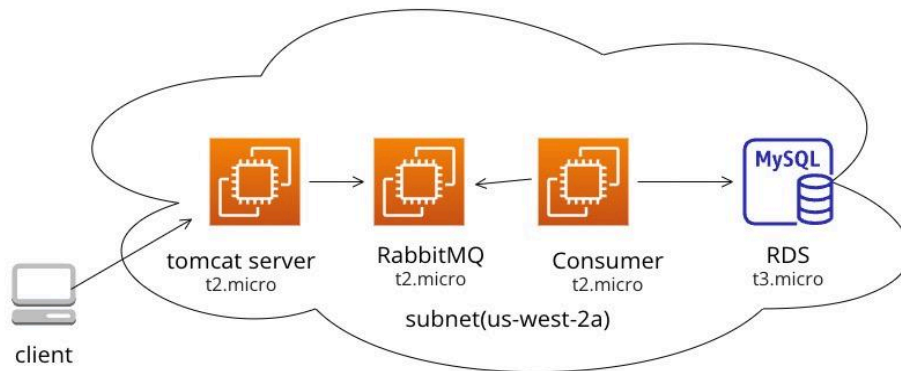
Database Design

I created a database named **skiers** in my AWS RDS instance. Within this database, a table named **lift_rides** is created to store data about each lift ride taken by skiers. By consolidating all relevant data into one table, I aim to improve performance and streamline data queries, ensuring faster access and simpler management.

Schema: lift_rides

Column Name	Data Type	Constraints	Description
id	INT	AUTO_INCREMENT, PRIMARY KEY	Unique identifier for each lift ride record.
skier_id	INT	NOT NULL	The ID of the skier taking the lift ride.
season_id	VARCHAR(10)	NOT NULL	The season during which the lift ride took place.
day_id	VARCHAR(10)	NOT NULL	The day of the season on which the lift ride occurred.
lift_id	INT	NOT NULL	The ID of the lift taken by the skier.
time	INT	NOT NULL	The time at which the lift ride started.
resort_id	INT	NOT NULL	The ID of the resort where the lift ride took place.

AWS Deployment Topologies



In a **single-AZ** deployment, I have deployed the consumer, broker, AWS RDS and server instances in the **us-west-2a** availability zone to ensure low latency. I choose the following instance type for balance of cost and performance

db.t3.micro: Part of the AWS free tier, used for AWS RDS.

t2.micro: Also part of the AWS free tier, used to run tomcat servers, message brokers, and consumers.

Ensuring High Throughput and Stability

1.Batch writing

To address the challenge of low throughput due to the write-heavy nature of processing each message individually, I introduced a batch write mechanism. This mechanism collects messages into batches of 1000 before writing them to the database, significantly improving efficiency and throughput.

2.circuit breaker

Given budget constraints, increasing the database or server capacity was not feasible. Instead, I implemented **throttling** to manage the load. This includes a throughput-based circuit breaker that monitors the system's performance and applies **exponential backoffs to client POST** requests when necessary. The circuit breaker helps to prevent system overload by temporarily halting or slowing down the processing of new requests, allowing the system to recover and maintain stable performance.

Client test run and RMQ console screenshots.

My best throughput was **1296 requests per second using 50 consumer threads and 200 client threads**, a MySQL connection pool size of 50, and 200 client threads. This configuration produced a stable queue length of 10 or less.

I ran tests with different numbers of consumer threads (50, 60, 100). The maximum connection pool size was limited to 60 due to the t3.micro instance's support for a maximum of 60 connections. Since the focus is on writing to the database, the client configuration was kept constant with 200 threads.

Detailed test results:

1. 50 consumer threads

Queue skier_post_queue



```
Number of new threads at 1st batch: 32
Number of new threads at 2nd batch: 200
Number of successful requests: 200000
Number of unsuccessful requests: 0
Total run time (wall time): 154.392 s
Total throughput (requests per second): 1295.4039069381834
Mean response time: 160.83641 ms
Median response time: 56.0 ms
99th percentile response time: 1849.0 ms
Minimum response time: 18 ms
Maximum response time: 23372 ms
```

2. 60 consumer threads

Queue skier_post_queue

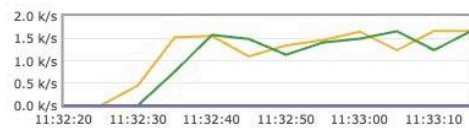
Overview

Queued messages [last minute](#) [?](#)



Ready	0
Unacked	1
Total	1

Message rates [last minute](#) [?](#)



Publish	1,665/s	Consumer ack	1,664/s
Deliver (manual ack)	1,662/s	Redelivered	0.00/s
Deliver (auto ack)	0.00/s	Get (manual ack)	0.00/s

Details

Features	queue storage version: 1	State	running	Messages	?
Policy		Consumers	60	Message body bytes	?
Operator policy		Consumer capacity	100%	Process memory	?
Effective policy definition					

```
Number of new threads at 1st batch: 32
Number of new threads at 2nd batch: 200
Number of successful requests: 200000
Number of unsuccessful requests: 0
Total run time (wall time): 161.199 s
Total throughput (requests per second): 1240.70248574743
Mean response time: 160.90628 ms
Median response time: 54.0 ms
99th percentile response time: 1944.0 ms
Minimum response time: 17 ms
Maximum response time: 23478 ms
```

3. 100 consumer threads


Queue skier_post_queue

▼ Overview

Queued messages

last minute

?



Ready

0

Unacked

0

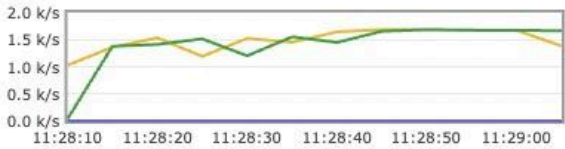
Total

0

Message rates

last minute

?



Publish

1,382/s

Deliver (manual ack)

1,664/s

Deliver (auto ack)

0.00/s

Consumer ack

1,664/s

Redelivered

0.00/s

Get (manual ack)

0.00/s

Details

Features

Policy

Operator policy

Effective policy definition

queue storage version: 1

State

running

Consumers

100

Consumer capacity

100%

Messages

?

Message body bytes

?

Process memory

?

```
Number of new threads at 1st batch: 32
Number of new threads at 2nd batch: 200
Number of successful requests: 200000
Number of unsuccessful requests: 0
Total run time (wall time): 198.814 s
Total throughput (requests per second): 1005.9653746718038
Mean response time: 213.39688 ms
Median response time: 59.0 ms
99th percentile response time: 2803.0 ms
Minimum response time: 18 ms
Maximum response time: 37420 ms
```