Assignment 3

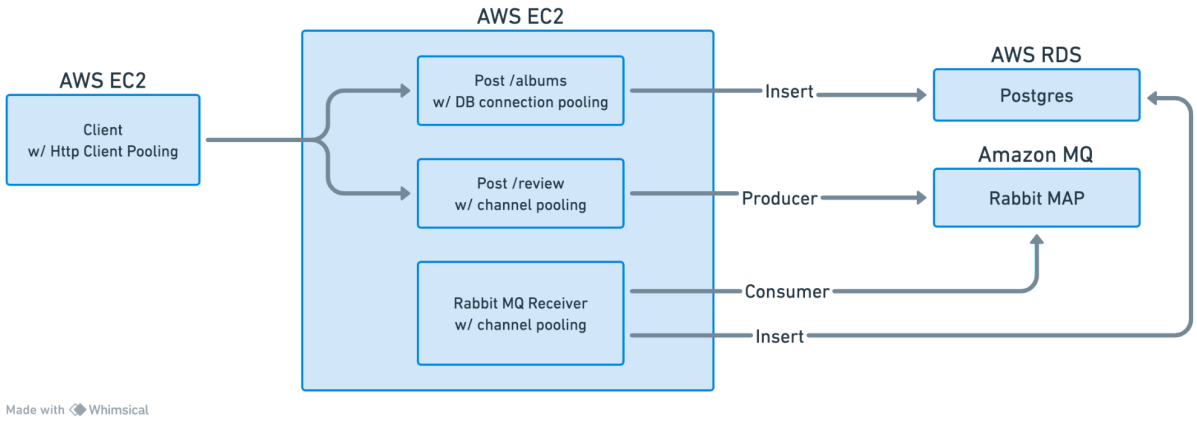
CS 6650, Fall 2023, Ximing Liang

GitHub Repository URL

https://github.com/ximing0116/CS6650-Assignment3

Description of your server design:

- Relationships:



- Components:

Servlets and the rabbitmq receiver run in an EC2 instance of **t2.micro** type.

Database uses **PostgreSQL** as the engine and **db.t3.micro** as the class.

Amazon MQ broker uses **rabbitmq** as the engine and **mq.t3.micro** as the instance type.

Client runs in an EC2 instance of **t2.micro** type.

- Major Classes:

ReviewServlet.java : implemented handling logics of post requests to /review/{likeornot}/{albumID}

RabbitMQReceiver.java: implemented consuming events from rabbitmq, parsing and inserting into the database.

AlbumServlet.java: implemented handling logics of get and post request to /albums.

SimpleClient.java: implemented the client that issues post requests to /review and /album in multithreaded fashion, and eventually prints out stats.

- Packages:

Packages are not explicitly leveraged in my project, but there are packages from external dependent libraries I can introduce here:

org.apache.http.*: used to create http client that reuses connection by pooling.
com.fasterxml.jackson.*: used to parse responses in Json format.
com.rabbitmq.client.*: used to initialize and communicate with rabbitmq server.
com.zaxxer.hikari.*: used to create a reusable pool of db connections.
java.sql.*: used to interact with databases.

- Messages Lifecycle:

Message is created inside ReviewServlet.java when a post request to /review is received. ReviewServlet first parsed the *like or not* and *albumID* parameters from the path, then composed a string message, and finally sent to rabbitmq and returned success. Instead of using a single channel, a pool of N channels is used to scale the throughput at producer side.

Messages sent to rabbitmq are pushed to RabbitMQReceiver. When messages are arrived, it parses the message and uses the parameters to compose the insert sql to insert into the database.

- Table Schema

```
 postgres=> \d album_likes
            Table "public.album_likes"
  Column  |  Type   | Collation | Nullable | Default
----------+---------+-----------+----------+---------
 albumid  | integer |           | not null |
 likes    | integer |           |          | 0
 dislikes | integer |           |          | 0
```

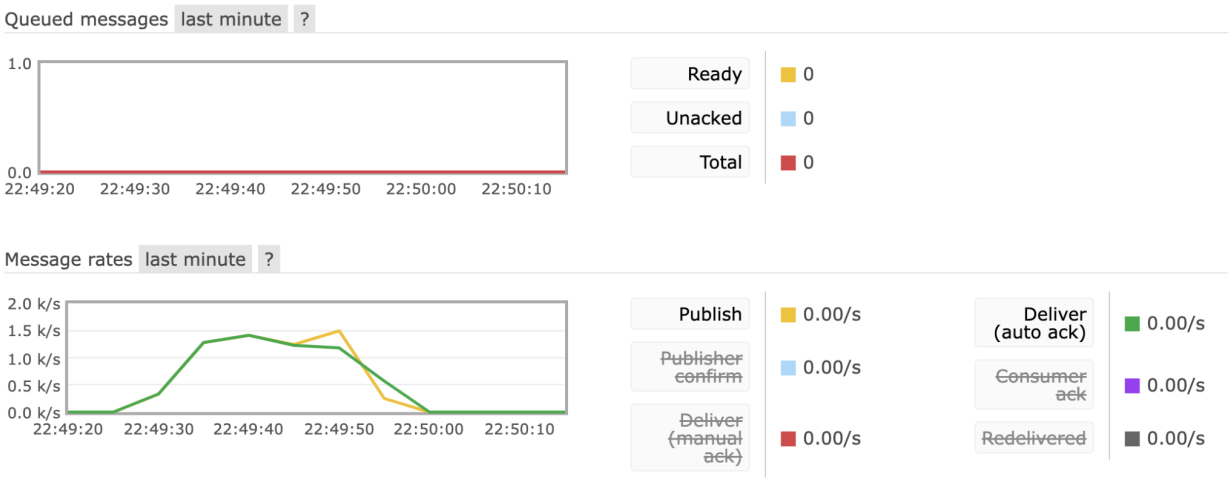
Test run results (command lines showing metrics, RMQ management windows showing queue size, send/receive rates) showing your best throughput:

Total requests handled: 10 * 10 * 100 * 4 = 40000, note 4 consists of 1 /albums + 3 /review requests

```
============================================================
Execution started at: 1701492216593
Execution ended at: 1701492236851
============================================================
Total execution time (sec): 20
Total throughput (reqs/sec): 1974
============================================================
Total successful total: 40000
Total failed non-200 total: 0
Total failed with exception total: 0
============================================================
============================================================
POST Stats:
Mean response time (ms): 6.5299
Median response time (ms): 2.0
p99 response time (ms): 40
Min response time (ms): 0
Max response time (ms): 477
```

RMQ management windows showing queue size, send/receive rates

Showing messages are consumed fast enough:



Showing messages are received and consumed around the same speed:

Overview					Messages			Message rates			+/-
Virtual host	**Name**	**Type**	Features	**State**	**Ready**	**Unacked**	**Total**	**incoming**	**deliver / get**	**ack**	
/	**review**	classic	**AWS-DEFAULT-POLICY-SINGLE-INSTANCE**	running	0	0	0	1,284/s	1,280/s	0.00/s	

Showing a pool of channels are used, in particular 2 channels are used for production and 10 channels for consumption.

Overview				Details			Message rates				+/-
Channel	**User name**	Mode ?	**State**	**Unconfirmed**	Prefetch ?	**Unacked**	**publish**	**confirm**	**unroutable (drop)**	**deliver / get**	**ack**
3.141.106.212:36818 (1)	root		running	0	1	0				136/s	0.00/s
3.141.106.212:36818 (10)	root		running	0	1	0				136/s	0.00/s
3.141.106.212:36818 (2)	root		running	0	1	0				136/s	0.00/s
3.141.106.212:36818 (3)	root		running	0	1	0				136/s	0.00/s
3.141.106.212:36818 (4)	root		running	0	1	0				136/s	0.00/s
3.141.106.212:36818 (5)	root		running	0	1	0				136/s	0.00/s
3.141.106.212:36818 (6)	root		running	0	1	0				136/s	0.00/s
3.141.106.212:36818 (7)	root		running	0	1	0				136/s	0.00/s
3.141.106.212:36818 (8)	root		running	0	1	0				136/s	0.00/s
3.141.106.212:36818 (9)	root		running	0	1	0				136/s	0.00/s
3.141.106.212:39908 (1)	root		running	0		0	663/s	0.00/s	0.00/s		
3.141.106.212:39908 (2)	root		running	0		0	686/s	0.00/s	0.00/s		

How many client threads are optimal to maximize system throughput?

If we just want to max throughput, using 256 threads provides the highest throughput at ~2800 RPS.

Thread	1	2	4	8	16	32	64	128	256	512
RPS	~250	~400	~700	~1000	~1400	~1800	~2000	~2400	~2800	~2400

Total request handled: 256 * 100 * 4 = 102400

```
==========================================================
Execution started at: 1701501418999
Execution ended at: 1701501456011
==========================================================
Total execution time (sec): 37
Total throughput (reqs/sec): 2766
==========================================================
Total successful total: 102400
Total failed non-200 total: 0
Total failed with exception total: 0
==========================================================
==========================================================
POST Stats:
Mean response time (ms): 75.1540625
Median response time (ms): 35.0
p99 response time (ms): 543
Min response time (ms): 0
Max response time (ms): 1916
```

Rabbitmq reached 2419 rps at one point, and delivery rate matched well with production rate.

Overview				Details			Message rates					+/-
Channel	**User name**	Mode ?	**State**	**Unconfirmed**	Prefetch ?	**Unacked**	**publish**	**confirm**	**unroutable (drop)**	**deliver / get**	**ack**	
3.141.106.212:54512 (1)	root		running	0		0	738/s	0.00/s	0.00/s			
3.141.106.212:54512 (2)	root		running	0		0	858/s	0.00/s	0.00/s			
3.141.106.212:54512 (3)	root		running	0		0	823/s	0.00/s	0.00/s			
3.141.106.212:55130 (1)	root		running	0	1	0				241/s	0.00/s	
3.141.106.212:55140 (1)	root		running	0	1	0				241/s	0.00/s	
3.141.106.212:55152 (1)	root		running	0	1	0				241/s	0.00/s	
3.141.106.212:55154 (1)	root		running	0	1	0				241/s	0.00/s	
3.141.106.212:55164 (1)	root		running	0	1	0				241/s	0.00/s	
3.141.106.212:55170 (1)	root		running	0	1	0				241/s	0.00/s	
3.141.106.212:55186 (1)	root		running	0	1	0				241/s	0.00/s	
3.141.106.212:55202 (1)	root		running	0	1	0				241/s	0.00/s	
3.141.106.212:55210 (1)	root		running	0	1	0				241/s	0.00/s	
3.141.106.212:55218 (1)	root		running	0	1	0				241/s	0.00/s	

How many queue consumers threads are needed to keep the queue size as close to zero as possible?

Production Side:
Connection Count 1
Channel Count 3

Consumer Side:
Connection Count 10
Channel Count 10 (1 channel per connection)

General rule of thumb based on my testing is to over-scale consumer than the producer in order to keep the queue size as low as possible, otherwise we will start to see the number of queued message to spike, like below: