Assignment 2 - Yang Chen

Git repo:

https://github.com/Yangyanggogo/SkiResortDistributedSystem/tree/hw3

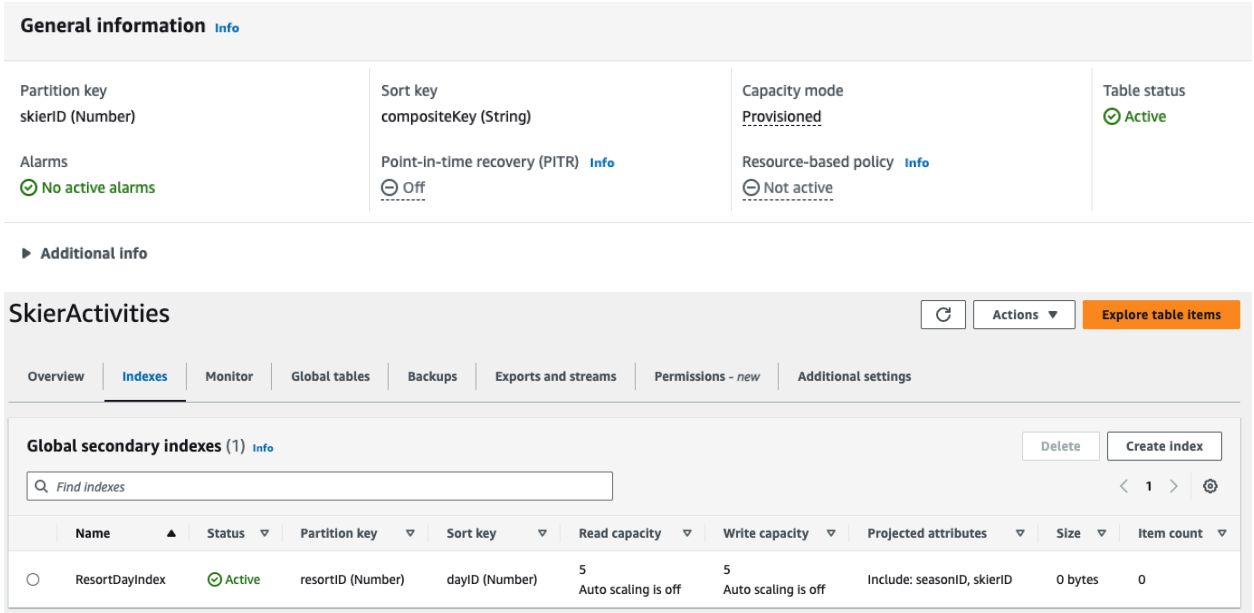
Database design

This assignment's main goal is to modify the consumer to persist the results in a database, and the database chosen for this assignment is DynamoDB since I expect very high write throughput and need to scale easily. DynamoDB is a NoSQL, scalable, serverless, good for unstructured data, and fast reads/writes.

DynamoDB table design

To be able to answer queries listed in Assignment 2 instructions, the table was designed as follows.

Table configuration with skierID as the partition key, compositeKey("seasons#dayID#time#liftID") as the sort key, and a Global Secondary Index (GSI) ResortDayIndex with resortID as the partition key and dayID as the sort key.



DynamoDB table access setting

Access to AWS services was ensured by using temporary credentials, specifically the access key ID, secret access key, and session token with AWS to enhance security. By manually obtaining temporary credentials from the AWS access portal and saving them in a local .aws/credentials file, the requests are authenticated to be written in the DynamoDB table created in AWS.

DynamoDB table create

To create a DynamoDB table in Java, I used the AWS SDK for Java. The Java code that creates a DynamoDB table suitable for storing the skier messages can be found in the git repo.

Queries answer example

The proposed answers for the listed 4 queries can be found in the queries.java file.

Redis database design

key: SkierId
value: seasonID|dayId|liftId

Data within Redis is managed using a hash structure keyed by a unique identifier, skierId. Each skierId is associated with a hash that stores multiple fields representing different dimensions such as seasonID, dayID, and liftID. These fields within the hash are updated to concatenate new incoming values, delimited by a pipe |, effectively accumulating a list of values within a single hash field. This method offers an efficient way to collate and access multi-dimensional data related to each unique skierId, allowing for quick updates and retrievals. The atomic operations used ensure that updates to the hash are consistent, and concurrent modifications by multiple clients are managed correctly by Redis's single-threaded processing model.

Redis access setting is ensured by the private IP address of the Redis EC2 instance because the consumer was deployed to EC2.

Deployment topologies on AWS

Server: Build artifacts of Server, then deploy into server's tomcat web app folder via tomcat management.

Client: run in local.

Consumer: scp Consumer.jar in EC2 instance and run it.
DynamoDB table: created on AWS.

Consumer class

The consumer class was updated to write messages in the DynamoDB table when polling out the messages in rabbitMQ.

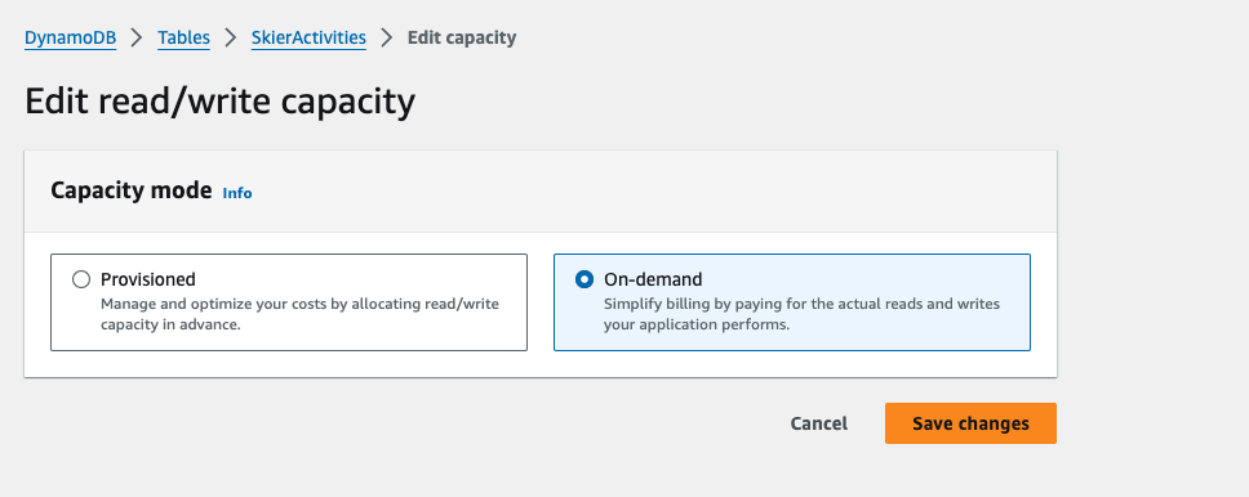
Writing performance improving

To improve my database's write performance and address write throttling, the following steps were considered.

- Permanently increase the provisioned WCUs for your table to handle higher loads. WCUs were set to be 50L in Creating the DynamoDB table Java code.

```
.provisionedThroughput(
    ProvisionedThroughput.builder()
        .readCapacityUnits( aLong: 5L)
        .writeCapacityUnits( aLong: 50L)
        .build()
)
.globalSecondaryIndexes(
    GlobalSecondaryIndex.builder()
        .indexName( s: "ResortDayIndex")
        .keySchema(
            KeySchemaElement.builder()
                .attributeName( s: "resortID")
                .keyType(KeyType.HASH)
                .build(),
            KeySchemaElement.builder()
                .attributeName( s: "dayID")
                .keyType(KeyType.RANGE)
                .build()
        )
        .projection(Projection.builder()
            .projectionType(ProjectionType.INCLUDE)
            .nonKeyAttributes( ...strings: "seasonID", "skierID")
            .build())
        .provisionedThroughput(
            ProvisionedThroughput.builder()
                .readCapacityUnits( aLong: 5L)
                .writeCapacityUnits( aLong: 50L)
                .build()
        )
```

- **Modify the DynamoDB table to on-demand mode.**



- Implement exponential backoff in your application's retry logic.

```
if (!requestItems.isEmpty()) {
  try {
    Thread.sleep( millis: (long) Math.pow(2, attempts) * baseBackoffTime);
  } catch (InterruptedException e) {
    Thread.currentThread().interrupt();
    return; // Early exit if the thread's interrupted status is set
  }
  attempts++;
}
```

- Use the `BatchWriteItem` API to perform batch writes to DynamoDB instead of individual `PutItem` requests, which can be more efficient.

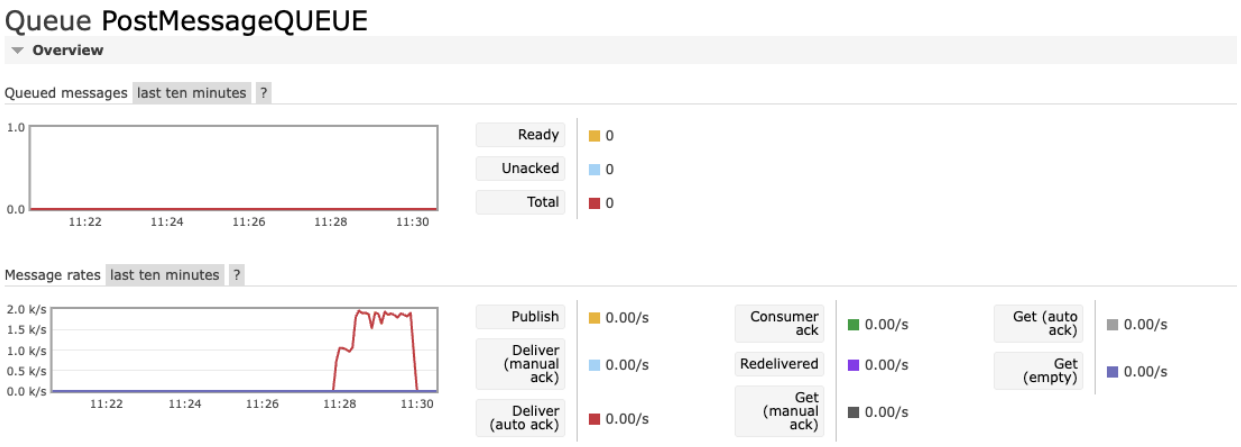
```
public class DynamoDBBatchWriter {

  2 usages
  private final DynamoDbClient dynamoDbClient;
  2 usages
  private final String tableName;
  6 usages
  private final List<WriteRequest> writeRequests;
  2 usages
  private final int batchSize;
  2 usages
  private final int maxRetryAttempts;
  2 usages
  private final int baseBackoffTime;
```

Details of implementations to improve writing performance can be found in the code.

Test results

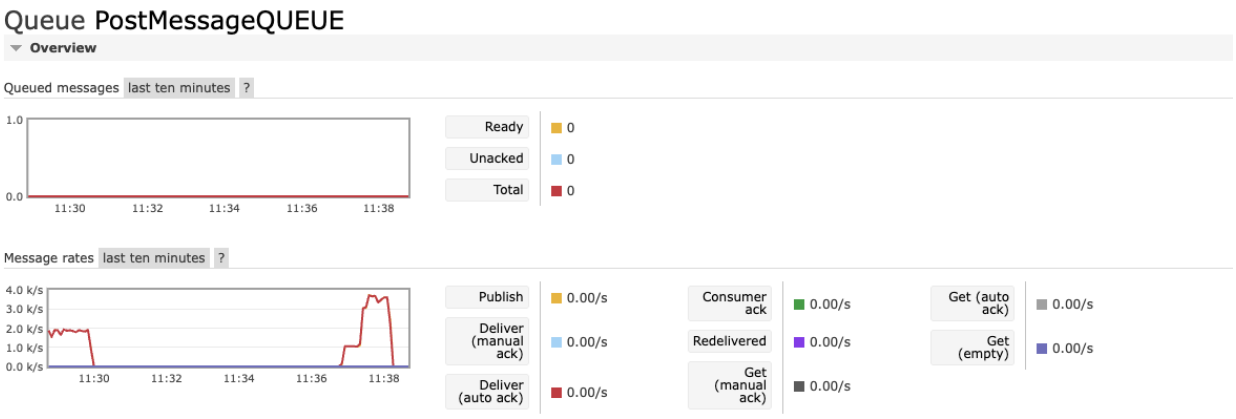
Client: 32 threads for 1000 requests in phase 1. 64 threads in phase 2;
Consumer: 200 threads

```
-------------------------------------
Multi threads consumer test start
-------------------------------------
Number of threads in phase 2: 64
Number of successful requests: 200000
Number of fail requests: 0
Wall Time: 122734
Mean latency= 34.183995 ms
Median latency= 33.0 ms
P99 latency= 71.0 ms
Throughout: 1639 requests/second
Min latency= 15
Max latency= 1055
-------------------------------------
Multi threads consumer test end
-------------------------------------
```



Client: 32 threads for 1000 requests in phase 1. 128 threads in phase 2;
Consumer: 200 threads

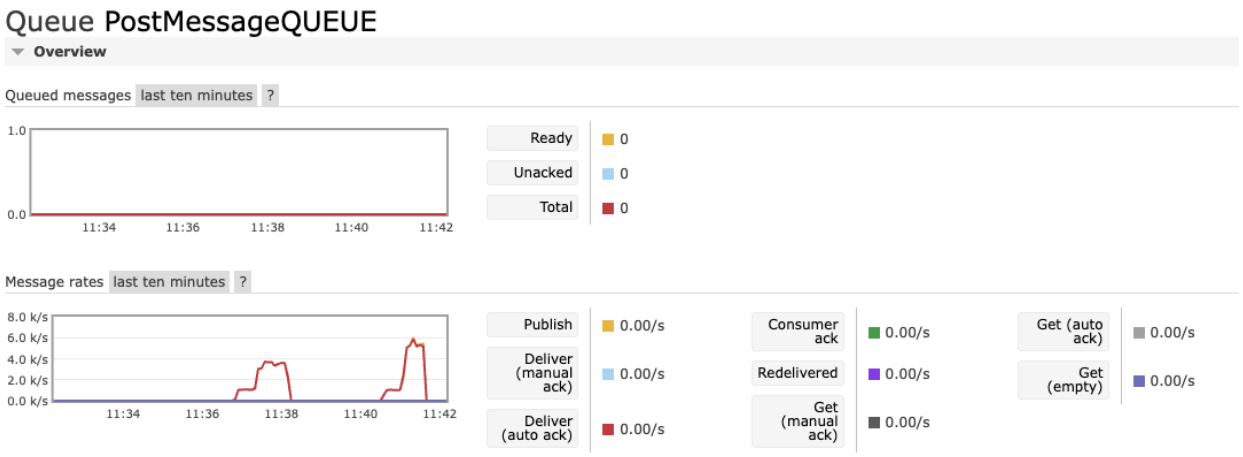
```
Number of threads in phase 2: 128
Number of successful requests: 200000
Number of fail requests: 0
Wall Time: 79390
Mean latency= 37.56143420638214 ms
Median latency= 34.0 ms
P99 latency= 67.0 ms
Throughout: 2531 requests/second
Min latency= 14
Max latency= 10166
-------------------------------------
Multi threads consumer test end
-------------------------------------
```



Client: 32 threads for 1000 requests in phase 1. 256 threads in phase 2;

Consumer: 200 threads

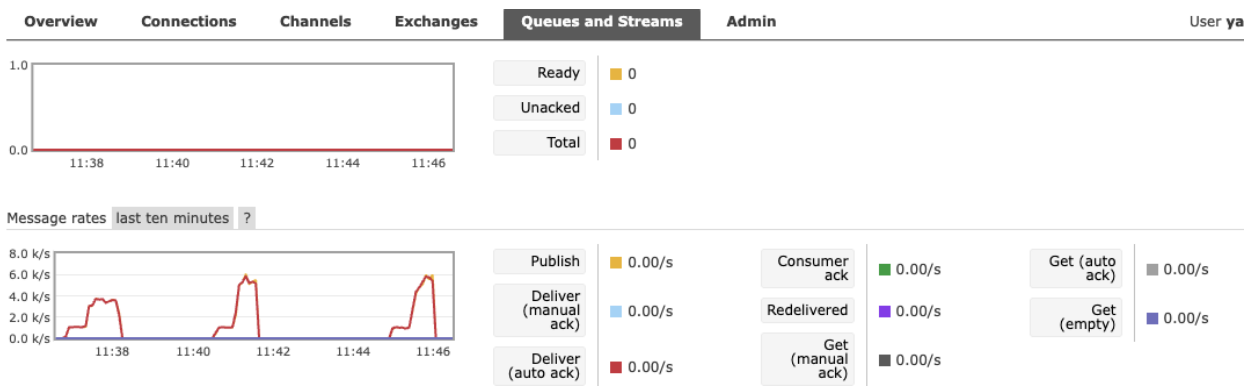
```
Number of threads in phase 2: 256
Number of successful requests: 200000
Number of fail requests: 0
Wall Time: 70537
Mean latency= 47.87553858303926 ms
Median latency= 37.0 ms
P99 latency= 103.0 ms
Throughout: 2857 requests/second
Min latency= 13
Max latency= 10154
-------------------------------------
Multi threads consumer test end
```



Client: 32 threads for 1000 requests in phase 1. 350 threads in phase 2;

Consumer: 200 threads

```
Number of threads in phase 2: 350
Number of successful requests: 200000
Number of fail requests: 0
Wall Time: 64985
Mean latency= 67.82839938034081 ms
Median latency= 47.0 ms
P99 latency= 196.0 ms
Throughout: 3125 requests/second
Min latency= 15
Max latency= 20411
-------------------------------------
Multi threads consumer test end
```



Client: 32 threads for 1000 requests in phase 1. 500 threads in phase 2;
Consumer: 200 threads

```
Number of threads in phase 2: 500
Number of successful requests: 200000
Number of fail requests: 0
Wall Time: 61391
Mean latency= 69.63689562979435 ms
Median latency= 61.0 ms
P99 latency= 186.0 ms
Throughout: 3278 requests/second
Min latency= 13
Max latency= 10318
-------------------------------------
Multi threads consumer test end
```

Client: 32 threads for 1000 requests in phase 1. 800 threads in phase 2;

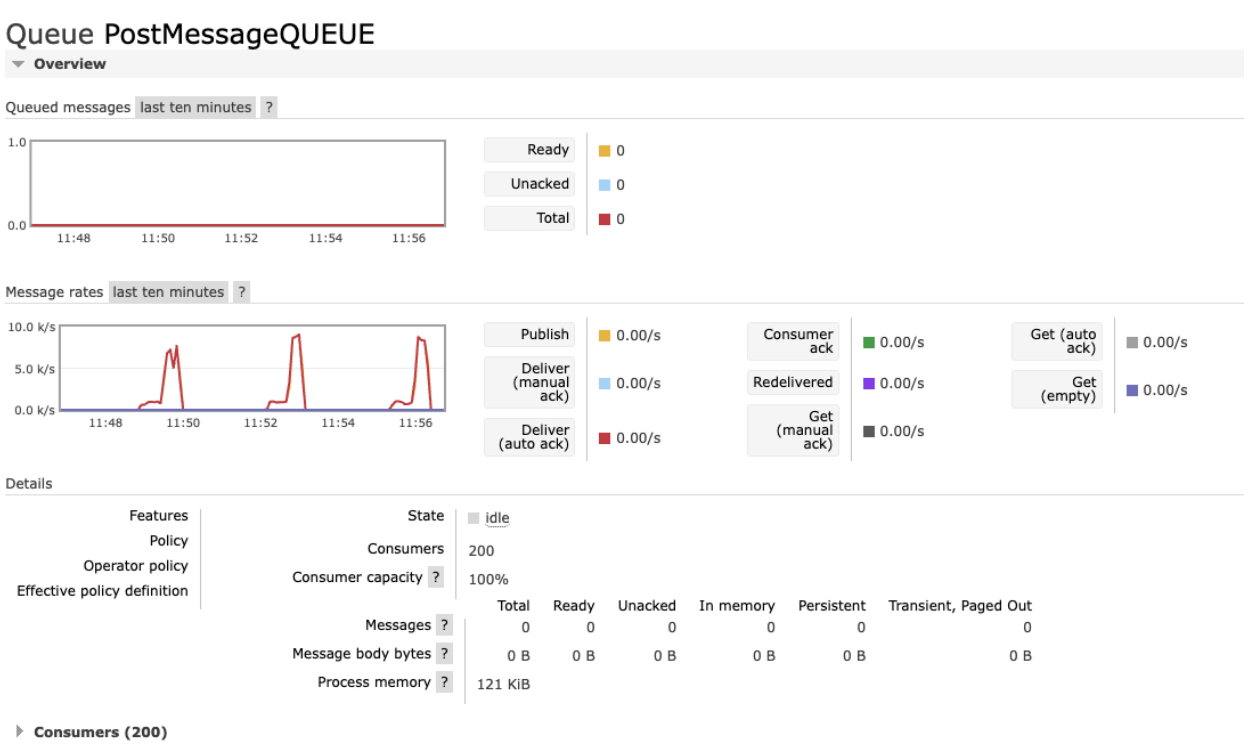
Consumer: 200 threads

```
Number of threads in phase 2: 800
Number of successful requests: 200000
Number of fail requests: 0
Wall Time: 53516
Mean latency= 88.44545899804086 ms
Median latency= 79.0 ms
P99 latency= 188.0 ms
Throughout: 3773 requests/second
Min latency= 15
Max latency= 10345
-------------------------------------
Multi threads consumer test end
```

Client: 32 threads for 1000 requests in phase 1. 1000 threads in phase 2;

Consumer: 200 threads

```
Number of threads in phase 2: 1000
Number of successful requests: 200000
Number of fail requests: 0
Wall Time: 57028
Mean latency= 110.24639752173479 ms
Median latency= 100.0 ms
P99 latency= 415.0 ms
Throughout: 3508 requests/second
Min latency= 15
Max latency= 10514
--------------------------------------
Multi threads consumer test end
```



In conclusion, according to the tests shown above, the best throughput is around 3800 requests/second with almost 0 messages stocked in the rabbitMQ, 800 threads in the phase 2 client part, and 200 threads in the consumer part. The following part talks more about the comparison of data writing in the DynamoDB table and Redis.

Data writing speed Table setting with writeCapacityUnits = 50 L

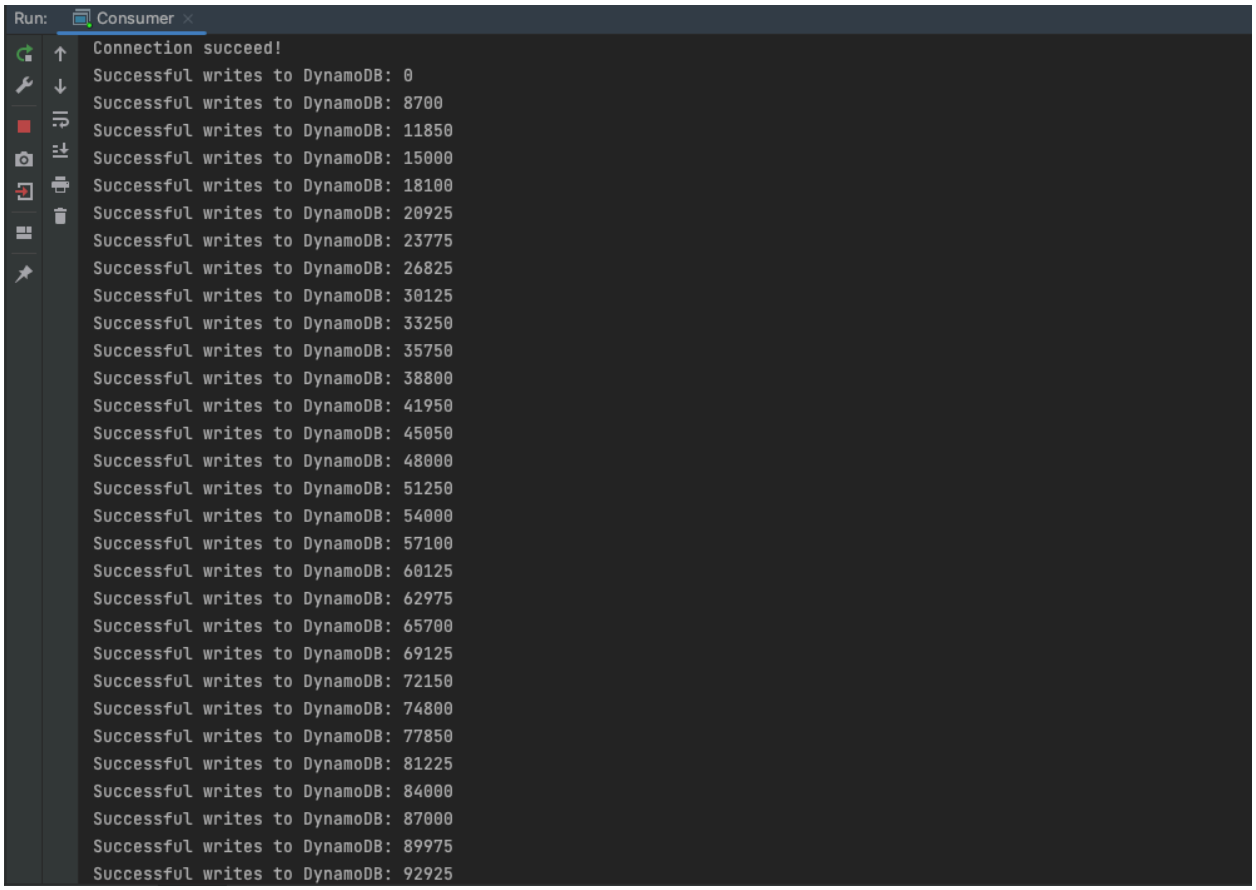
Client: 32 threads for 1000 requests in phase 1. 800 threads in phase 2;

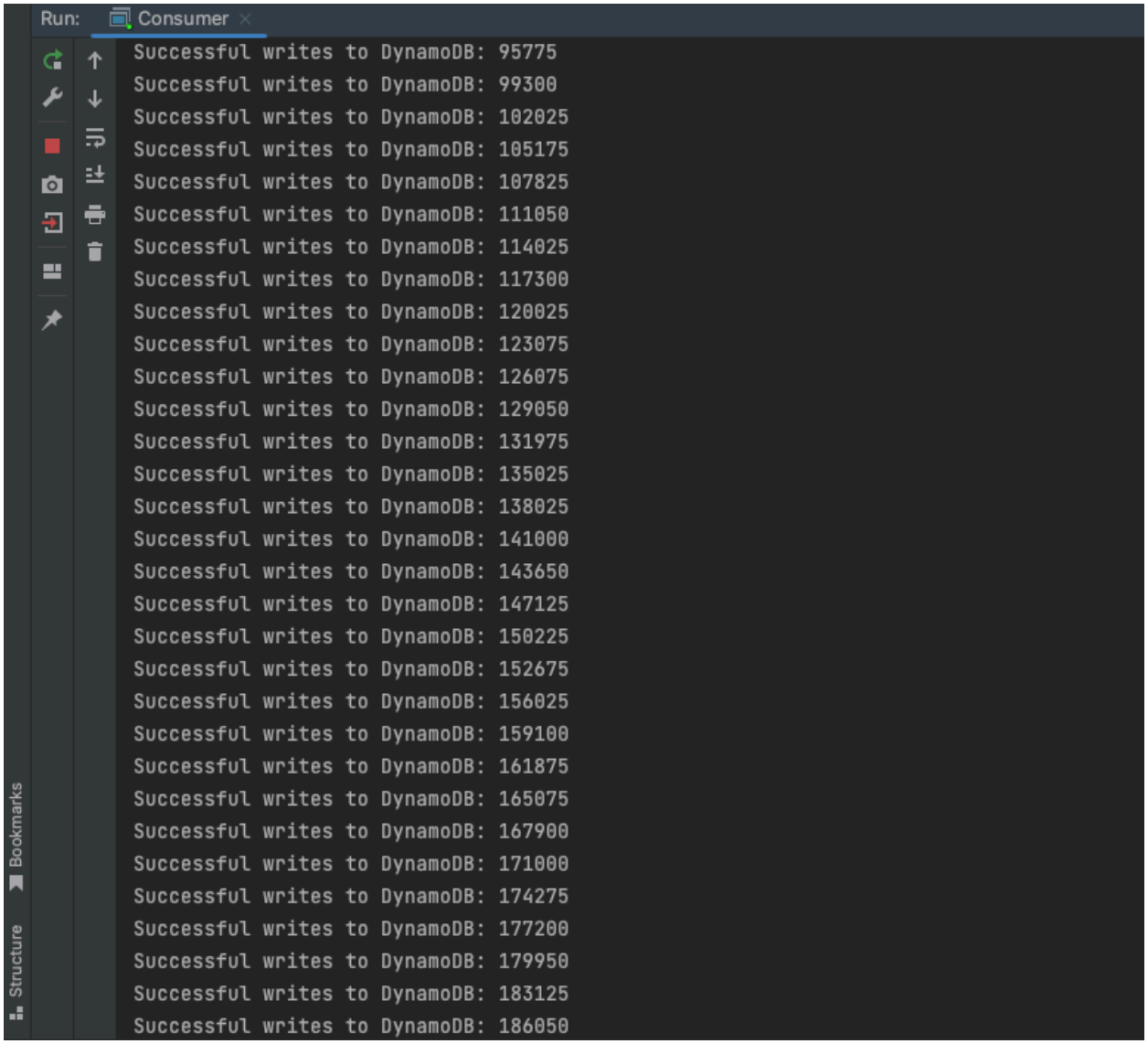
Consumer: 200 threads

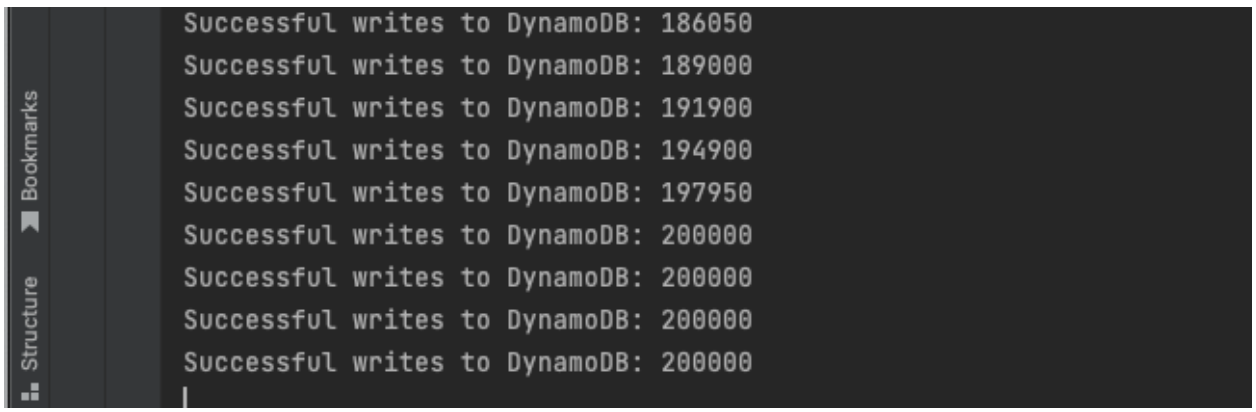
It took around 65 minutes to finish writing in dynamoDB.

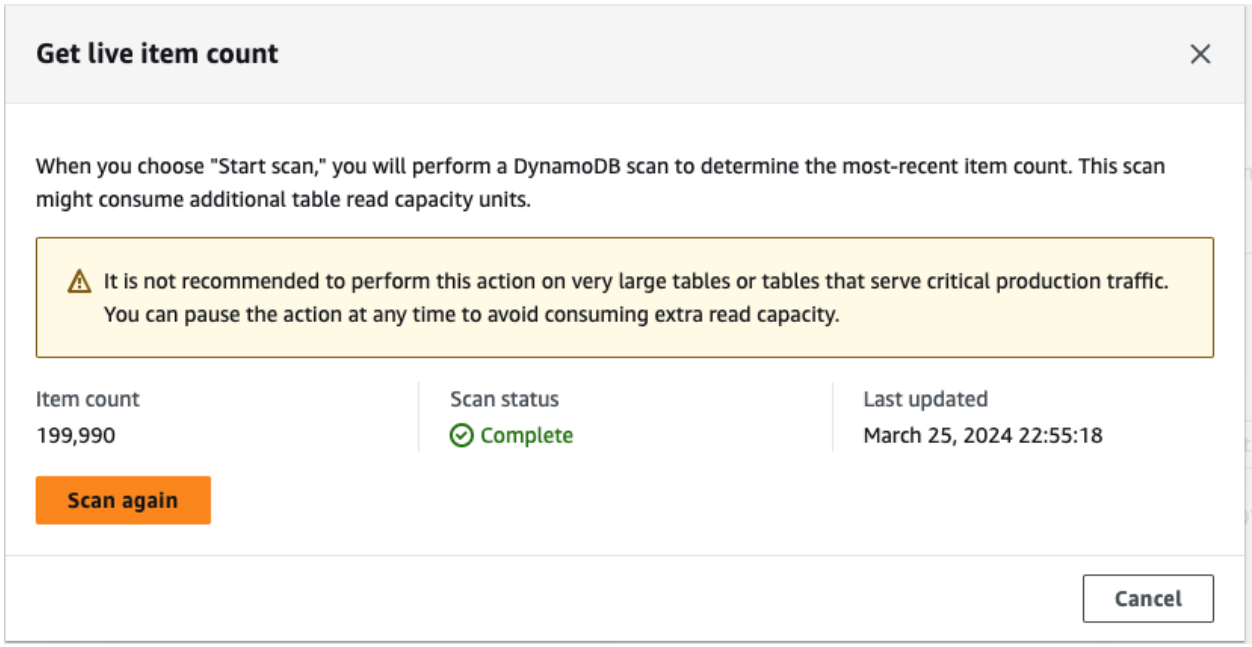
```
Number of threads in phase 2: 800
Number of successful requests: 200000
Number of fail requests: 0
Wall Time: 54051
Mean latency= 99.73182111083915 ms
Median latency= 92.0 ms
P99 latency= 239.0 ms
Throughout: 3703 requests/second
Min latency= 14
Max latency= 10650
-------------------------------------
Multi threads consumer test end
-------------------------------------
```

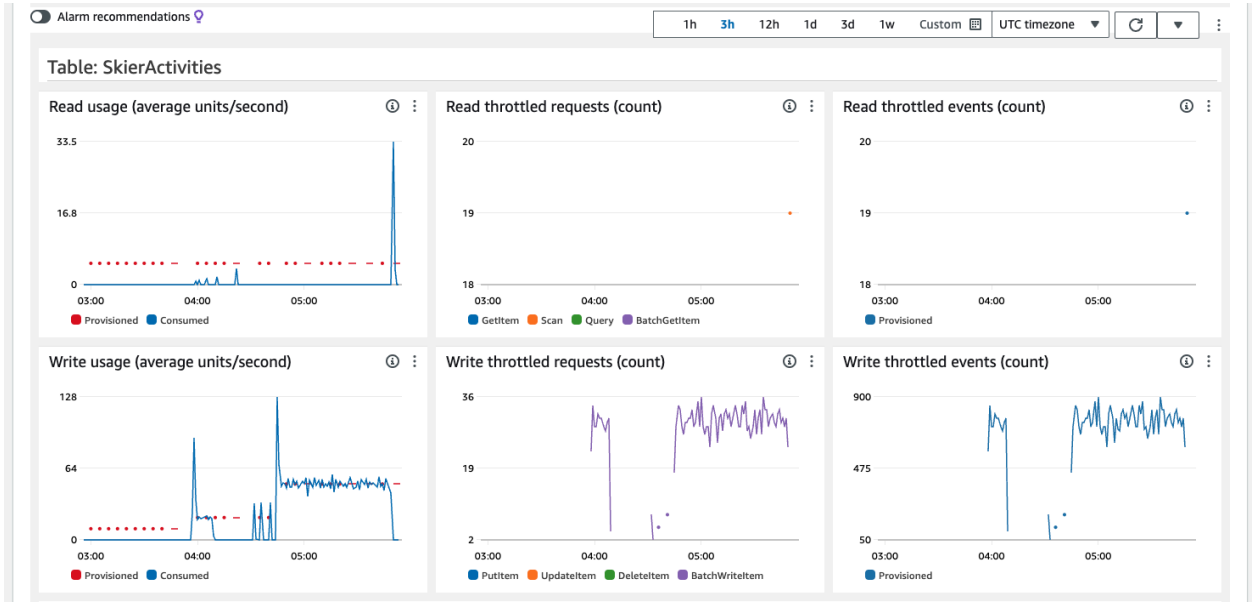
DynamoDB table items




Run:
Consumer
Successful writes to DynamoDB: 95775
Successful writes to DynamoDB: 99300
Successful writes to DynamoDB: 102025
Successful writes to DynamoDB: 105175
Successful writes to DynamoDB: 107825
Successful writes to DynamoDB: 111050
Successful writes to DynamoDB: 114025
Successful writes to DynamoDB: 117300
Successful writes to DynamoDB: 120025
Successful writes to DynamoDB: 123075
Successful writes to DynamoDB: 126075
Successful writes to DynamoDB: 129050
Successful writes to DynamoDB: 131975
Successful writes to DynamoDB: 135025
Successful writes to DynamoDB: 138025
Successful writes to DynamoDB: 141000
Successful writes to DynamoDB: 143650
Successful writes to DynamoDB: 147125
Successful writes to DynamoDB: 150225
Successful writes to DynamoDB: 152675
Successful writes to DynamoDB: 156025
Successful writes to DynamoDB: 159100
Successful writes to DynamoDB: 161875
Successful writes to DynamoDB: 165075
Successful writes to DynamoDB: 167900
Successful writes to DynamoDB: 171000
Successful writes to DynamoDB: 174275
Successful writes to DynamoDB: 177200
Successful writes to DynamoDB: 179950
Successful writes to DynamoDB: 183125
Successful writes to DynamoDB: 186050
Bookmarks
Structure


Successful writes to DynamoDB: 186050
Successful writes to DynamoDB: 189000
Successful writes to DynamoDB: 191900
Successful writes to DynamoDB: 194900
Successful writes to DynamoDB: 197950
Successful writes to DynamoDB: 200000
Successful writes to DynamoDB: 200000
Successful writes to DynamoDB: 200000
Successful writes to DynamoDB: 200000
Bookmarks
Structure

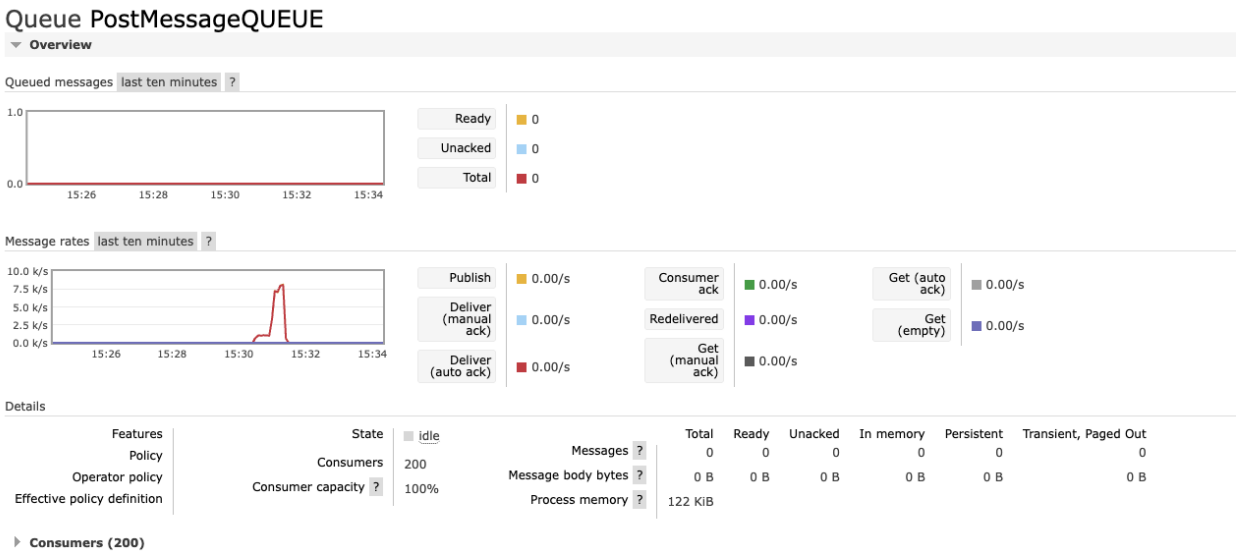

Get live item count
When you choose "Start scan," you will perform a DynamoDB scan to determine the most-recent item count. This scan might consume additional table read capacity units.
It is not recommended to perform this action on very large tables or tables that serve critical production traffic. You can pause the action at any time to avoid consuming extra read capacity.
Item count
199,990
Scan status
Complete
Last updated
March 25, 2024 22:55:18
Scan again
Cancel


Alarm recommendations
1h 3h 12h 1d 3d 1w Custom
UTC timezone
Table: SkierActivities
Read usage (average units/second)
Read throttled requests (count)
Read throttled events (count)
Write usage (average units/second)
Write throttled requests (count)
Write throttled events (count)
Provisioned
Consumed
GetItem
Scan
Query
BatchGetItem
PutItem
UpdateItem
DeleteItem
BatchWriteItem

DynamoDB Data with On-demand

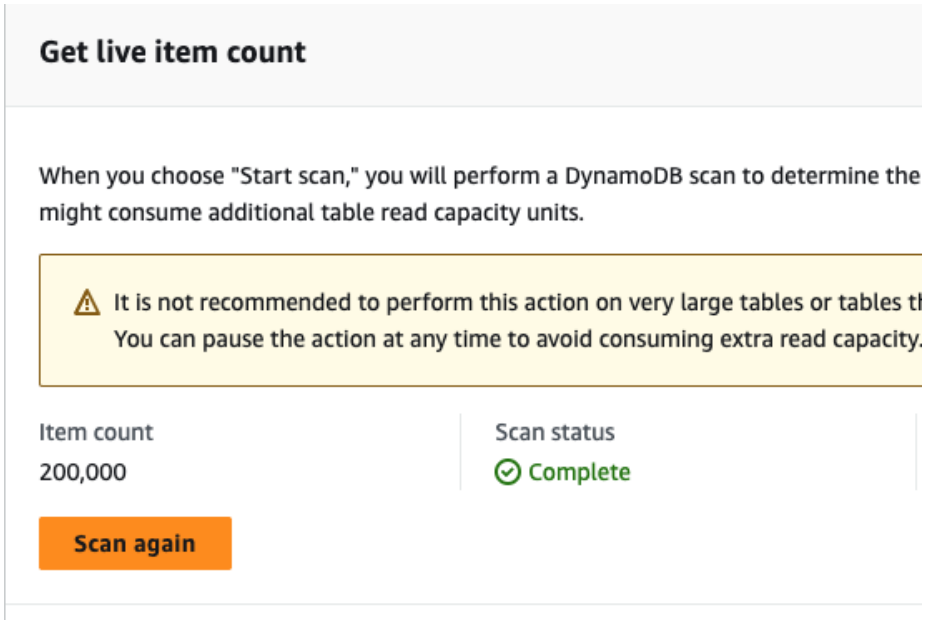
It took around 3 minutes to finish writing in DynamoDB.

```
Number of threads in phase 2: 800
Number of successful requests: 200000
Number of fail requests: 0
Wall Time: 63412
Mean latency= 89.35119818086409 ms
Median latency= 82.0 ms
P99 latency= 168.0 ms
Throughout: 3174 requests/second
Min latency= 15
Max latency= 10349
```




```
[ec2-user@ip-172-31-29-95 ~]$ java -jar /opt/Consumer.jar
Connecting...
log4j:WARN No appenders could be found for logger (com.rabbitmq.client.impl.Cons
umerWorkService).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more in
fo.
Connection succeed!
Successful writes to DynamoDB: 0
Successful writes to DynamoDB: 44575
Successful writes to DynamoDB: 145175
Successful writes to DynamoDB: 200100
Successful writes to DynamoDB: 200100
```

(* the extra 100 writing was caused by the previous test with 100 writing.)


Get live item count
When you choose "Start scan," you will perform a DynamoDB scan to determine the
might consume additional table read capacity units.
It is not recommended to perform this action on very large tables or tables t
You can pause the action at any time to avoid consuming extra read capacity.
Item count
200,000
Scan status
Complete
Scan again

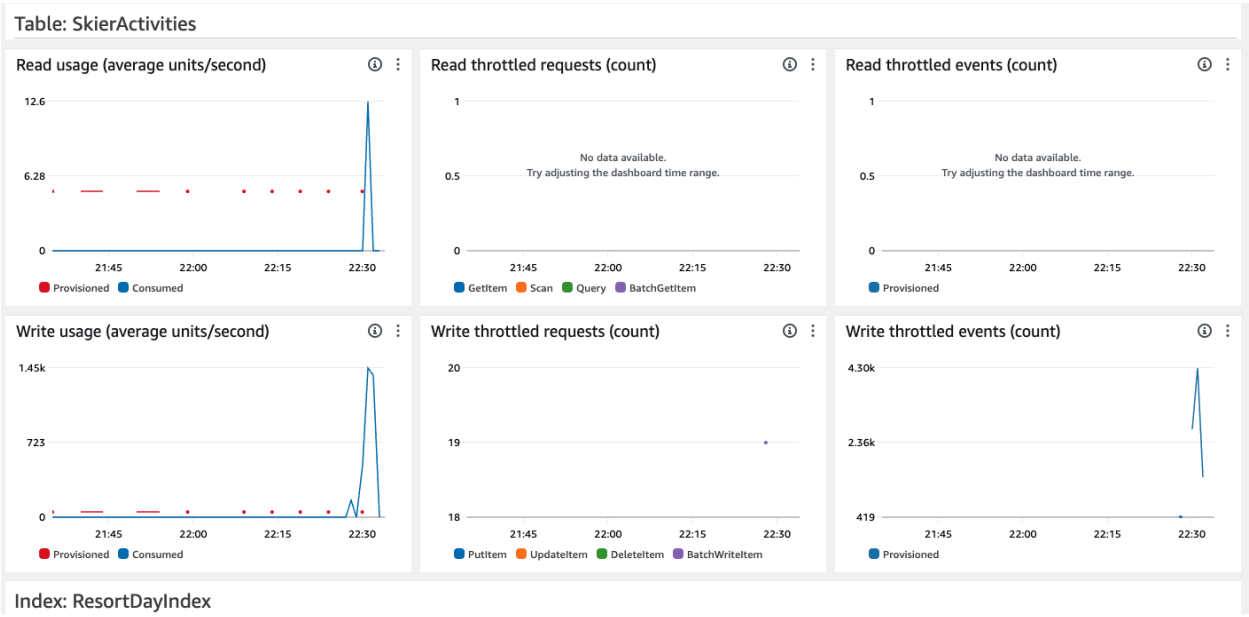
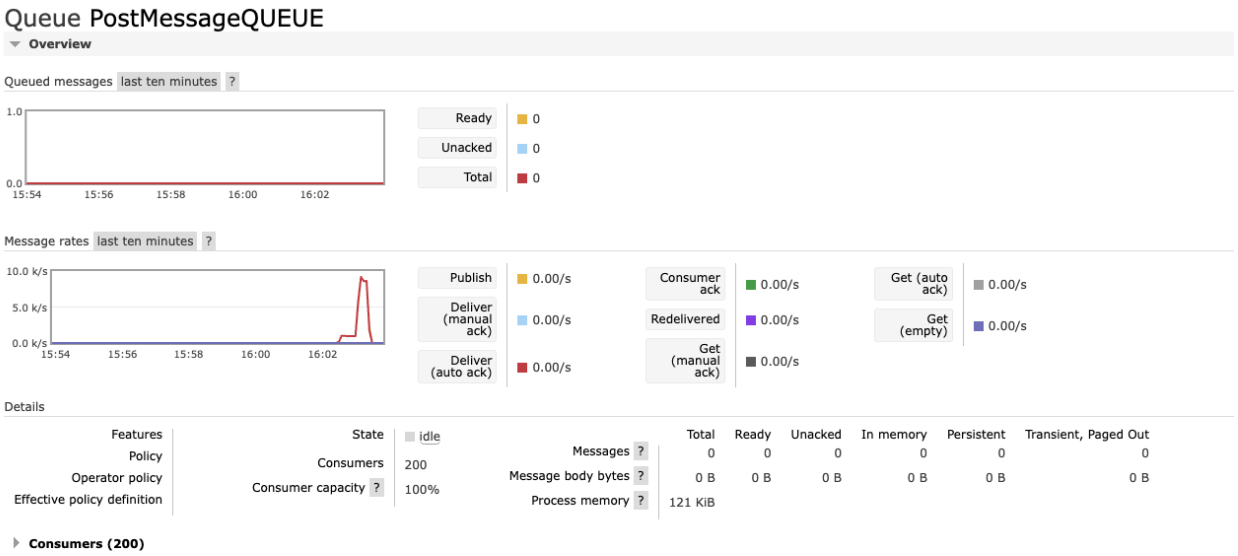


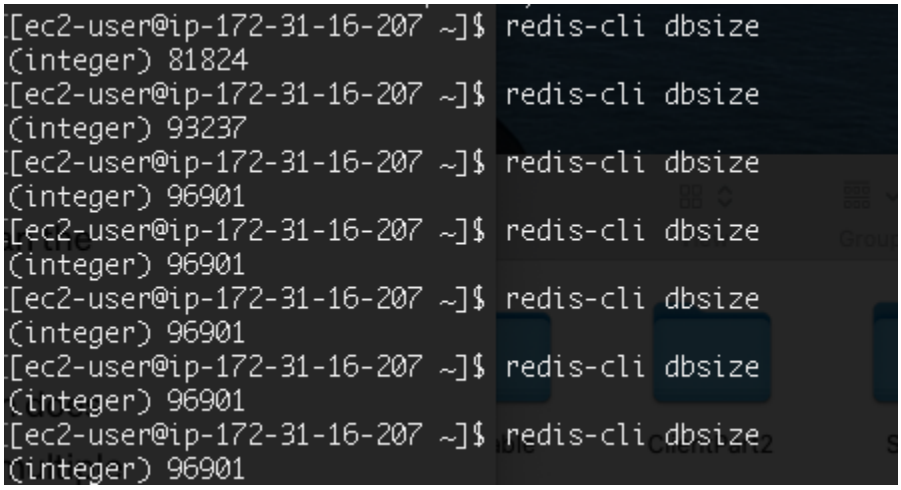
Table: SkierActivities
Read usage (average units/second)
12.6
6.28
0
21:45
22:00
22:15
22:30
Provisioned
Consumed
Read throttled requests (count)
1
0.5
0
No data available.
Try adjusting the dashboard time range.
21:45
22:00
22:15
22:30
GetItem
Scan
Query
BatchGetItem
Read throttled events (count)
1
0.5
0
No data available.
Try adjusting the dashboard time range.
21:45
22:00
22:15
22:30
Provisioned
Write usage (average units/second)
1.45k
723
0
21:45
22:00
22:15
22:30
Provisioned
Consumed
Write throttled requests (count)
20
19
18
21:45
22:00
22:15
22:30
PutItem
UpdateItem
DeleteItem
BatchWriteItem
Write throttled events (count)
4.30k
2.36k
419
21:45
22:00
22:15
22:30
Provisioned
Index: ResortDayIndex

Redis Writing result

It took around 3 minutes to finish writing in Redis. The throughput is slightly greater than writing in DynamoDB.

```
Number of threads in phase 2: 800
Number of successful requests: 200000
Number of fail requests: 0
Wall Time: 52773
Mean latency= 86.46589966314485 ms
Median latency= 77.0 ms
P99 latency= 200.0 ms
Throughout: 3846 requests/second
Min latency= 14
Max latency= 10488
-------------------------------------
Multi threads consumer test end
```





The Redis database size is less than 200k because it is stored by skierId, which is in the range of 1 to 100000. Every new record will update the value of the keys.

Test with Circuit Breaker

Although in my previous chapters, the tests show that there is no backlogs in the message queue, a new server was implemented using a circuit breaker.

In this part of the report, I introduce throttling and a circuit breaker with exponential backoffs in the context of a client application interacting with a message queue (RabbitMQ) and a database (DynamoDB and Redis).

To implement a circuit breaker on the server side, I utilize the EventCountCircuitBreaker from Apache Commons Lang, which opens the circuit after certain events occur (such as failed attempts to write to the database). The implementation details are in my GitHub codebase, in the Server3 file.

All tests shown below share the same parameters for consumer and client:
800 threads in client phase 2; 200 threads in consumer.

DynamoDB Data writing setting with On-demand

```
Number of threads in phase 2: 800
Number of successful requests: 200000
Number of fail requests: 0
Wall Time: 50899
Mean latency= 83.73784195667925 ms
Median latency= 81.0 ms
P99 latency= 188.0 ms
Throughout: 4000 requests/second
Min latency= 14
Max latency= 11018
-------------------------------------
Multi threads consumer test end
```

```
[ec2-user@ip-172-31-29-95 ~]$ java -jar /opt/Consumer.jar
Connecting...
log4j:WARN No appenders could be found for logger (com.rabbitmq.client.impl.Cons
umerWorkService).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more in
fo.
Connection succeed!
Successful writes to DynamoDB: 0
Successful writes to DynamoDB: 49675
Successful writes to DynamoDB: 150600
Successful writes to DynamoDB: 200050
```

Get live item count

When you choose "Start scan," you will perform a DynamoDB scan to determine the most-recent item count. This scan might consume additional table read capacity units.

⚠ It is not recommended to perform this action on very large tables or tables that serve critical production traffic. You can pause the action at any time to avoid consuming extra read capacity.

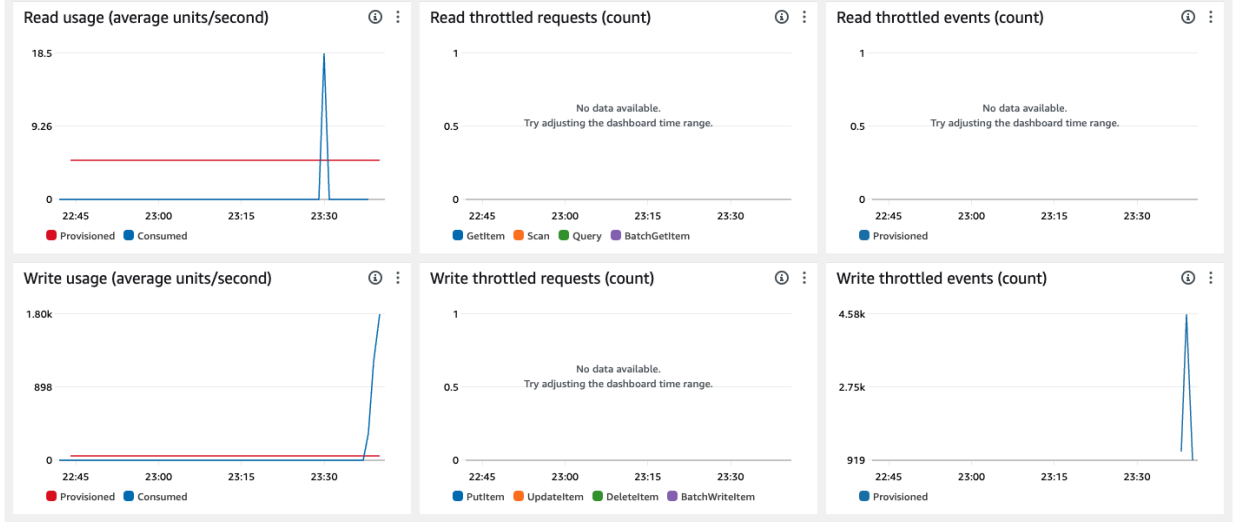
Item count	Scan status	Last updated
200,000	Complete	March 26, 2024 17:57:21

Scan again

Cancel

Table capacity metrics View all metrics

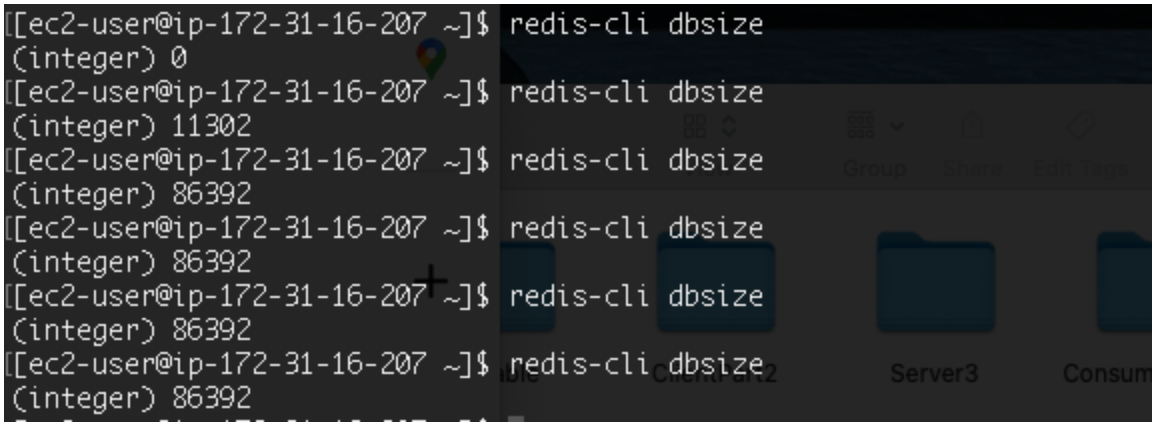
Alarm recommendations 1h 3h 12h 1d 3d 1w Custom UTC timezone



Redis Writing result

```
Number of threads in phase 2: 800
Number of successful requests: 200000
Number of fail requests: 0
Wall Time: 51260
Mean latency= 84.62066345783789 ms
Median latency= 78.0 ms
P99 latency= 192.0 ms
Throughout: 3921 requests/second
Min latency= 14
Max latency= 10396
-------------------------------------
Multi threads consumer test end
```


```
[ec2-user@ip-172-31-29-95 ~]$ java -jar /opt/Consumer_redis.jar
Connecting...
log4j:WARN No appenders could be found for logger (com.rabbitmq.client.impl.Con
umerWorkService).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more i
fo.
Connection succeed!
Successful Redis writes: 0
Successful Redis writes: 41056
Successful Redis writes: 119521
Successful Redis writes: 200002
```



Conclusion

Best throughput

The best throughput is 4000 requests/second, with 800 threads in the client phase 2 and 200 threads in the Consumer.

Message queue

All tests with a reasonable number of threads in the Consumer ensure that there are no backlog messages in the queue.

Database choice

Using DynamoDB with an on-demand setting has the same writing speed as Redis.