Assignment 3

**GitHub Repo:**

https://github.com/xq443/distributed-system/tree/main/Assignment/3

**Redis DB Design:**

1.Redis Key Design

Examples

- Skier Data
  - Key: skier:{skierID}:season:{seasonID}
  - Value: Hash with fields like:
    - days_skiied (Set of dayIDs skied).
    - vertical:{dayID} (Vertical total for the day).
    - lifts:{dayID} (List of lifts ridden).
- Resort Visitors
  - Key: resort:{resortID}:day:{dayID}:visitors
  - Value: Set of skierIDs.

2.API Retrieval Logic

API Endpoint:

GET /skier/{skierID}/season/{seasonID}/summary

Steps for Data Retrieval:

1. Retrieve Days Skied
   Query: SCARD skier:{skierID}:season:{seasonID}:days_skiied
   - Efficiently counts unique skiing days.
2. Retrieve Vertical Totals
   Query: HGETALL skier:{skierID}:season:{seasonID}:vertical
   - Returns all day-to-vertical mappings for the skier.
3. Retrieve Lifts by Day
   For each dayID: Query: LRANGE skier:{skierID}:season:{seasonID}:lifts:{dayID} 0 -1
   - Returns the list of lifts ridden on a specific day.
4. Response Construction
   Combine the results into a JSON response summarizing days skied, total vertical feet, and lifts ridden per day.

3. Optimal Key Design Trade-offs

Chosen Key Structure

- □ Example: skier:{skierID}:season:{seasonID}:vertical
  - Pros:
    - Optimized for skier-centric queries (e.g., vertical totals per day).
    - Prevents unnecessary data fetching by scoping to the skier and season.
  - Cons:
    - Separate keys for vertical and lifts can require multiple queries for full retrieval.

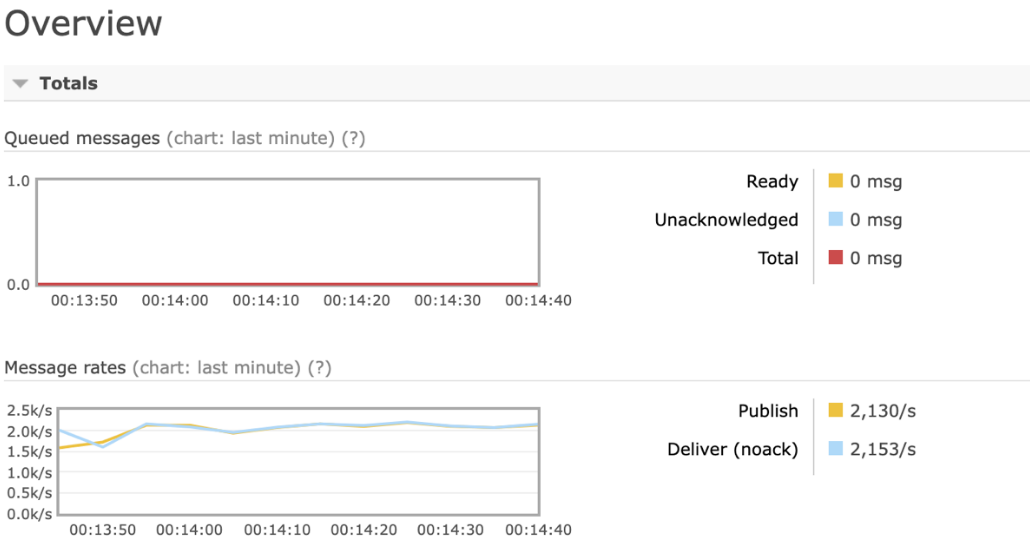
Alternative Design

- □ Key: skier:{skierID}:season:{seasonID}:day:{dayID}
    - Pros:
        - Each day's data (vertical and lifts) is stored together for simpler retrieval.
        - Fewer Redis keys overall.
    - Cons:
        - Requires fetching and aggregating all day:{dayID} keys for season-level queries.
        - Increases query complexity for aggregating totals.

4. Trade-offs between Redis and other database choices
Redis is less suited for complex queries or durability but excels in low-latency, high-throughput use cases. If real-time performance is the priority, Redis is the optimal choice, while DynamoDB or MySQL would be better for large-scale durability or relational needs.

**Deployment Topology on AWS:**
The deployment is composed of an EC2 instance and redis within AWS.
Redis resides in the same EC2 instance with consumer application that reads messages from RabbitMQ; Tomcat server and RabbitMQ resides in another EC2 instance correspondingly.

**Throughput Performance:**
Instance type
z1d.large * 3



Overview

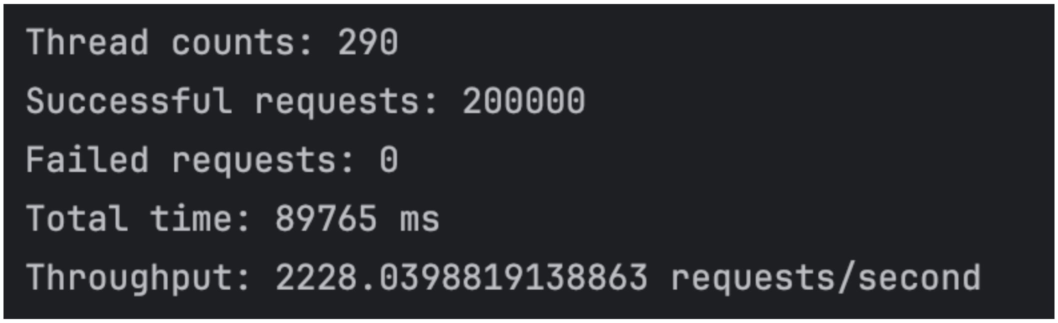
Totals

Queued messages (chart: last minute) (?)

| | |
|---|---|
| Ready | 🟨 0 msg |
| Unacknowledged | 🟦 0 msg |
| Total | 🟥 0 msg |

Message rates (chart: last minute) (?)

| | |
|---|---|
| Publish | 🟨 2,130/s |
| Deliver (noack) | 🟦 2,153/s |



```
Thread counts: 290
Successful requests: 200000
Failed requests: 0
Total time: 89765 ms
Throughput: 2228.0398819138863 requests/second
```