Assignment4 Design

Proposed Architecture Design

Given your requirements and the changes, here's a high-level architecture that includes read/write splitting, Redis, and MongoDB:

1. **Client Servers (Clients)**: Generate and send both POST and GET requests. Clients can be web or mobile applications.

   ### Potential Architecture with Four Client Servers:

   - **Client Server 1 (POST /LiftRide)**: Handles generating synthetic data for lift rides and sending POST requests.
     - 现有客户端
   - **Client Server 2 (GET /Skiers)**: Handles retrieving skier-specific data.
     - GET/skiers/{resortID}/seasons/{seasonID}/days/{dayID}/skiers/{skierID}
   - **Client Server 3 (GET /Resorts)**: Handles retrieving resort-specific data.
     - GET/resorts/{resortID}/seasons/{seasonID}/day/{dayID}/skiers
   - **Client Server 4 (GET /Vertical)**: Handles retrieving skier vertical data.
     - GET/skiers/{skierID}/vertical

   For GET request client server, do the followings:

   - Handle concurrency, rate limiting, and connection pooling for interactions with the backend.
   - get result back from backend and do some operations to see the results

2. **Load Balancer**: Distributes incoming requests to different instances of Skier Services and Resort Services, ensuring even load distribution and high availability.
3. **Backend Server**

   a. Divide into two micro-services:

   - **Skier Services**:

- Handles POST requests to write data related to skier activities.
- Handles GET requests for skier-specific data.
- Interacts with both Redis (for caching) and RabbitMQ - MongoDB (for persistent storage).

- **Resort Services**:
  - Handles GET requests related to resort data.
  - Like Skier Services, it can interact with Redis and MongoDB as necessary.

For both services:

Caching Strategy:

- Implement caching using Redis to store and quickly retrieve frequently accessed data. This can dramatically reduce the load on your database and speed up response times for these queries.
- Decide on a caching strategy (e.g., write-through, write-behind, cache-aside) that best fits your use case.

For this method, need to reconfigure the AWS loadbalancer to make sure it send our the correct server based on different requests.

b. all-in-one server:
   i. handles all requests(build up corresponding servlet)
   ii. for GET requests(cache-aside)
      1. first read from Redis, if the data is read, then return back to the client server
      2. if the data is unread, then build connection to RabbitMQ and further read from MongoDB
      3. write the data from MongoDB to Redis
   iii. for POST requests
      i. build connection to RabbitMQ and further write into MongoDB

4. **Redis (Cache Layer)**:

- Provides fast access to frequently read data, reducing load on MongoDB.
- Ideal for read-heavy operations like `GET/skiers/{skierID}/vertical` where data does not change frequently.

5. **MongoDB (Database Layer)**:
   - Stores persistent data.
   - Handles write operations from POST requests and read operations not served by Redis.
6. **Message Queue (RabbitMQ/Kafka)**:
   - Decouples write operations from direct database writes, improving throughput and reliability.
   - Skier Services post messages to the queue, which are then consumed by workers to write data into MongoDB.
7. Message Queue Consumer:
   - Consume messages from the Message Queue.
   - Perform database write operations asynchronously.

Visualization

```
[Client Servers] --(GET/POST)--> [Load Balancer] --(Distribute)--

                                                          (Read/Wr
                                                          [Redis C

                                                        (Write)
                                                          [Message
```

Redis and MongoDB Integration

- **Redis** is used as a caching layer to store hot data, such as frequently accessed skier verticals or resort information. This reduces the need for repetitive reads from MongoDB, speeding up response times for these queries.
- **MongoDB** serves as the primary data store for all persistent data, including skier activity logs and resort details. Write operations are queued in the Message Queue to allow for asynchronous processing, reducing the load on the database and improving overall system throughput.

Using Both Redis and MongoDB

- **Read and Write Splitting**: Use Redis for handling read-heavy operations by caching frequently accessed data. MongoDB handles all persistent storage needs, including writes and reads not covered by the cache.
- **Caching Strategy**: Implement caching policies in Redis, such as Least Recently Used (LRU) for cache eviction, to ensure that only the most relevant data is cached.
- **Data Consistency**: Implement mechanisms to ensure consistency between Redis and MongoDB. For example, when an update occurs in MongoDB, invalidate the corresponding cache in Redis.