Analysis Report on Twitter Redis Application Performance

Hardware Configuration

Operating System: Mac (Macbook M1 Pro 2020)
Number of Cores: 8
RAM: 16 GB

Software Stack

Database: Redis
Programming Language: Python 3.9.13
Main libraries Used:

- redis-py 4.6.0
- csv
- datetime
- random

Set-Up Instructions

For set-up instructions, please navigate to README.md in the folder submitted. In this file, you will have access to prerequisites, installation steps, environment configurations, and how to run the application. For any questions, kindly contact me at gollapudi.j@northeastern.edu.

Performance Testing Results

The performance was evaluated using two different strategies for managing tweets and timelines in Redis:

Strategy 1 (S1) - Simple set operation for posting tweets, on-the-fly timeline construction.

Average Results from 3 Iterations of testings:

S1: Posting Tweets Performance (Tweets Posted Per Second)

- Iteration 1: **6615.06** [1,000,000/151.17s]

- Iteration 2: **6428.80** [1,000,000/155.55s]
- Iteration 3: **6828.73** [1,000,000/146.44s]
- Average: **6624.19**

S1: Fetching Home Timelines Performance (Timelines Retrieved Per Second)

- Iteration 1: **0.01** [2/193.78s]
- Iteration 2: **0.01** [2/172.96s]
- Iteration 3: **0.01** [2/172.32s]
- Average: **0.01**

Strategy 2 (S2) - Immediate timeline updates upon posting tweets for faster retrieval.

Average Results from 3 Iterations of testings:

S2: Posting Tweets Performance (Tweets Posted Per Second)

- Iteration 1: **2542.52** [1,000,000/393.31s]
- Iteration 2: **2567.72** [1,000,000/389.45s]
- Iteration 3: **2608.71** [1,000,000/383.33s]
- Average: **2572.98**

S2: Fetching Home Timelines Performance (Timelines Retrieved Per Second)

- Iteration 1: **925.92** [1000/1.08s]
- Iteration 2: **934.57** [1000/1.07s]
- Iteration 3: **917.43** [1000/1.09s]
- Average: **925.97**

<u>Factors Affecting Performance</u>

1. Data Retrieval Method:

   In S1, the home timeline is constructed dynamically by fetching tweets from all users that a particular user follows. This process involves a more complex data retrieval method that scans through a potentially large dataset to find relevant tweets. The use of the SCAN command in Redis to iterate over keys matching a pattern (tweet_s1:*) can become less efficient as the number of tweets increases, leading to longer retrieval times.

2. Redis SCAN Command Efficiency:

   The SCAN command is designed to incrementally iterate over keys in the Redis database. While it prevents blocking the server for a long time (which commands like KEYS might do), it is not optimized for speed when dealing with large datasets. The need to repeatedly call SCAN until all relevant keys are found can significantly impact performance, especially when the dataset grows.

3. Complexity of Data Aggregation:

   After fetching the relevant tweets using SCAN, S1 requires additional steps to filter these tweets based on follower relationships and then sort them to construct the timeline. These operations add computational overhead, further affecting the retrieval speed. The need to deserialize tweet data, check if the tweet belongs to a followed user, and then order tweets based on timestamps increases the complexity of the retrieval process.

4. Network Latency and Server Load:

   While likely less significant in this context since the database is accessed locally, network latency and server load can still impact performance. In environments where the Redis server is accessed over a network, network delays can exacerbate retrieval times. Additionally, high server load can affect the responsiveness of the Redis server, further slowing down operations.

Analysis

- The testing demonstrates the trade-offs between the two strategies: Strategy 1 excels in posting speed but lags in timeline retrieval, while Strategy 2 offers a more balanced performance.

- These findings highlight the importance of choosing the right strategy based on application requirements, whether it's faster write operations or quicker read access.

- The application's performance underscores the scalability and efficiency of Redis as a database for handling high-volume, real-time data operations, although the choice of strategy significantly impacts its efficiency in specific tasks.

Conclusion

- The comparative analysis of S1 and S2 within the context of a Redis-based Twitter-like application highlights critical insights into data structure design and retrieval strategies in NoSQL databases. S1, while conceptually simpler and adhering to the assignment's dynamic retrieval requirement, exhibits performance challenges that limit its practical applicability in high-demand environments. The long retrieval times observed for S1 indicate a bottleneck that could lead to user dissatisfaction in real-world applications, where speed and efficiency are paramount.

- On the other hand, S2's approach to precomputing and storing timelines represents a more viable strategy for social media platforms. However, this efficiency comes at the cost of increased complexity in tweet posting and potential concerns over data freshness and storage efficiency.

- In conclusion, this performance analysis illustrates the importance of choosing the appropriate data management strategy based on application requirements, expected load, and user experience expectations. For future implementations, I will consider strategies such as data partitioning or advanced caching mechanisms to optimize performance and scalability further.