Sidney La Fontaine and Hari Muralikrishnan

Twitter Performance with Redis:

RATES	Strategy 1	Strategy 2
postTweet	**16,667 tweets/sec**	**19.3 tweets/sec**
getHomeTimeline	**0.12 requests/sec**	**6.7 requests/sec**

Abstract:

In this project report, we analyze the difference between using a relational and non-relational database to handle the operations of twitter, such as posting a tweet and retrieving a home timeline. Below we go in-depth on our use of Java to create an API to interact with our Redis database, a key-value store. We contrast this implementation and performance to our use of MySQL in assignment 1. In the end, we conclude that using a non-relational model, such as Redis, is preferable to using a relational model, MySQL, in this use case.

Introduction:

Last assignment, we used a relational model using MySQL to implement functionalities of Twitter including posting a million tweets, and randomly requesting home timelines for random users. In the process of doing so, we encountered problems with hardware and software capabilities, and we attributed the latter to the weaknesses of the relational model. In this assignment, we aim to see how our performances in these functions changed with a non-relational model in Redis.

Method:

We used Redis on Mac, and we implemented our database using specified key prefixes for tweets, followers, followees, and timelines. For our tweets, we stored our keys as "tweet:*:*", where the user_id is the first * and the tweet_id is the *. The value was stored in the entire tweet object in its string format. For our followers, we stored our keys as "followers:*", where the * is the user_id of the twitter user being followed. The value is the set of all twitter users following that key user_id. The opposite logic applies to followees. In other words, for our followees, we stored our keys as "followees:*", where the * is the user_id of a twitter user. The value is the set of all twitter users that this key user_id follows. Lastly, the home timeline was stored with the key "timeline:*" where the * is the user_id that this hometimeline belongs to, and the value is a list of tweet objects in their entirety representing the timeline.

We implemented a method called insertFollower, which would take in a Follower object (which has a user_id and a follows_id) and adds those values to the Redis database into both the followers and followees buckets. Our postTweet method took in a Tweet object and a boolean strategy (false = strategy 1, true = strategy 2). For both strategies, we inserted the key and value for that tweet in the style mentioned in the previous paragraph. For strategy 2, we additionally added this tweet into the timelines of all the people who follow the user posting this tweet. Lastly, we implemented getHomeTimeline which takes in a user_id and the same boolean strategy. For strategy 1, the method would have to find all the followees of this user, collect all the tweets posted by these followees, order them by date, and return the 10 most recent tweets. For strategy 2, it merely has to return the timeline object for this user id, sort by date, and return the 10 most recent, as the timeline has already been constructed as tweets were posted. Note that we took the recommendation of the professor and implemented helper methods including getFollowers, getFollowees, and getTweets.

Analysis:

First, let's analyze our performance rates for the postTweet method. For strategy 1, it was incredibly quick as expected because inserting key-values into a key-value store like Redis is exceptionally quick. For strategy 2, it was significantly slower because it involved getting all the followers for the poster of this tweet and then adding this tweet to a list for each of those followers. Strategy 1 was about 1,600 times faster at simply inserting tweets into their primary location (table in MySQL, key-value in Redis) using MySQL. Strategy 2 about 2 times faster. Therefore, we can conclude that for write purposes, we should rely on Redis.

Our performance for the getHomeTimeline method using Redis was also significantly better than it was when we user MySQL. For strategy 1, which requires much more work to retrieve a home timeline, our performance was still 10 times quicker using Redis than it was using MySQL. Although retrieving 0.12 home timelines a second is still slow, which can be attributed to the numerous steps involves, such as retrieving the tweets for all the user's followees and ordering them. Strategy 2, on the other hand, was much quicker, retrieving 6.7 home timelines per second. This was about 600 times quicker than retrieving home timelines using MySQL, which is a significant improvement. This can be attributed to doing extra work of updating each users' (who follows the user posting the tweet) timeline as tweets are being posted. Although it is worth pointing out that inserting the tweets with strategy 2 was about 800 times slower than with strategy 1. Meaning that strategy 2 would only be effective if we have much more home timeline requests than tweets being posted, which is the case for the reality of twitter.

Conclusions:

Our initial suspicion about each strategy having its own advantages was true. If we were developing twitter and debating between these strategies it would entirely depend on which operation speed we want to optimize, posting a tweet or retrieving a home timeline. Strategy 1 favors posting tweets, while strategy 2 favors retrieving home timelines. In reality, twitter only has to post about 6,000 to 10,000 tweets a second and handles 200 to 300 thousand home timeline requests per second. Therefore, we would favor strategy 2 due to the high volume of home timeline requests in comparison to volume of tweets being posted. Although even using Redis and strategy 2, our implementation would not be able to handle the tweets being posted or home timelines being requested per second. Nevertheless, it is clear that using a non-relational database, such as Redis, for such a large data engineering project is preferable to using a relational database, such as MySQL, as Redis outperformed MySQL in every category.

Author Contributions:

We both believe in working collaboratively, so we did all of our work together and in person. We used the pair programming technique to work on our project together, including the analysis.