Sara Adra, Anika Das
DS4300 HW2 Analysis

Results
Strategy 1 (optional)

API Method	API Calls/Sec
postTweet	8,531.34
getHomeTimeline	0.6196

Strategy 2 (required)

API Method	API Calls/Sec
postTweet	3,823.93
getHomeTimeline	1,096.79

Hardware Configurations and Software Stack:
Below, we have information regarding our hardware configuration and our software stack.

CPU Speed: 1.4 GHz Quad-Core Intel Core i5
Number of Cores: 4
RAM: 8GB
Disk: Capacity: 121.02 GB, Available: 9.7 GB (6.6 MB purgeable)

RDB (and version): MySQL8.0
Programming Language: Java
Libraries Used: Java.util, Java.sql, Java.time, Java.io

Implementation:

Due to how we structured our homework 1 assignment, we did not have to change very much in our main driver program which performs the timing analysis. In this second homework assignment, we are calling the same Twitter API methods, but they are now backed by a Redis implementation.

In terms of our main driver program, with the switch to redis/jedis, we no longer had to authenticate our access to the server which was previously done using a url, username, and password. We were able to remove the lines of code that retrieve the url, username, and password from our main driver file.

In implementing our Redis model (for strategy 2), we used lists to create buckets for (1) a tweet and all of its information (tweet id, user id, tweet timestamp, and tweet text), (2) for all of the tweet ids of tweets posted by a user, (3) for all of the user ids of users that follow a certain user, (4) for all of the user ids of users that a certain user follows, and (5) for all of the tweet ids of tweets on a user's timeline.

> For example, a tweet with an id of 1 would have a redis list named **tweet_1** with the following values ["1", "316", "2022-02-17 22:30:00", "hrv l fnj wmh skpy jqrrq urbxx drva rkxuhicvxo qrckpbackur k ly iuthsj dm xcdwi tgaypsu s"] - where "1" is the id of the tweet, "316" is the id of the user that posted this tweet, "2022-02-17 22:30:00" is the timestamp of the tweet, and "hrv l fnj wmh skpy jqrrq urbxx drva rkxuhicvxo qrckpbackur k ly iuthsj dm xcdwi tgaypsu s" is the text content of the tweet.
>
> For a user with an id of 1, we would also have a redis list labeled **user_1_tweets** with the following values ["998900", "995626", "991444", "988150", …] representing the ids of the tweets that this user (user_1) has posted.
>
> For this user, we would also have a redis list labeled **followers_1** with the following values ["6908", "5042", "7128", "4079", "6630"] representing the ids of users that follow this user (user_1).
>
> Further, we would have a redis list labeled **followees_1** with the following values ["9268", "8203", "5312"] representing the ids of users that this user (user_1) follows.
>
> We would also have a redis list labeled **timeline_1** with the following values ["998761", "997386", "984784", "979323", …] representing the ids of tweets posted by users that this user (user_1) follows.

In implementing the optional Strategy 1 Redis model, we changed our structure so that when we posted a tweet, we just used a set operation where the key was the tweet_id and the value was a string containing all of the contents/attributes of the tweet.
With this implementation, we still had our follower_id and followees_id buckets as redis lists, but we did not have a timeline_id bucket and our tweets were not in a redis list, but instead were a key-value object (with the key being tweet_id, and the value being "user_id;tweet_timestamp;tweet_text" (where the variables were filled in with their respective values)).
Therefore, because we did not have a timeline bucket that was constantly being updated with the tweets (in the form of tweet ids) on the user's timeline as with Strategy 2, when creating the timeline for a user, we had to construct the timelines on the spot.

Note: To load in the follower/followee data from the follows.csv file, we created a separate Java file (UploadFollowersData.java) where we iterated through the data in the file and for each user_id, follows_id row in the table, created instances of the two relationships (X follows Y, and Y is followed by X) in the respective followers_id and followees_id redis list buckets (where 'id' was replaced with the respective id value).

Analysis

After reading and understanding the two possible strategies for implementing the postTweet and getTimeline operations, we began with the second strategy as it was what the Twitter engineers ultimately decided to use. After implementing strategy 2 in which we copied the tweet id to the user's home timeline automatically after posting each tweet, the write performance was slower (3,823.93 tweets posted per second), while retrieving each timeline was much faster (1,096.79 timelines retrieved per second). As you can see, posting the tweets was much slower compared to strategy 1, but retrieving the home timelines was much faster when compared to strategy 1. This happened because the home timeline was already ready with all of the tweet_ids, all we had to do was go through these tweet_ids and create the respective Tweet objects.

We then implemented strategy 1 which resulted in a much faster time in terms of posting tweets (8,531.34 tweets posted per second); however, retrieving the home timelines took a lot longer (0.6196 timelines retrieved per second). As you can see, posting the tweets was considerably faster when compared to strategy 2, but retrieving the home timelines was a lot slower. This is most likely due to the fact that we went through every single tweet, rather than only the tweets of users followed by the given user - where, to find the most recent tweets with Strategy 1, we started with the most recent tweet (that with the "latest_tweet_id") and checked if the user who posted the tweet was followed by the user (who we are getting the timeline for) - if so, we added the tweet to the list of Tweets, and if not, we decremented the "latest_tweet_id" and looked at the next most recent tweet. We ran through this process until we either (1) collected the ten most recent tweets on the user's timeline, or (2) ran out of tweets to look through.

Partner Code Documentation

Throughout this homework assignment, we both worked on everything together through in-person work sessions and Zoom calls with screen-sharing. For example, we talked through our code implementation as Sara coded the programs on her computer through Zoom screen-sharing, and we walked through the process of connecting to Redis, running the code, and calculating the rates on Anika's computer as she shared her screen over Zoom.