DS4300 HW2
Jason Zhang and Marco Tortolani

DESIGN CHOICES

1. Implemented both Strategy 1 and Strategy 2
2. For Strategy 1, when tweets are posted, we add a reference to the tweet (id) to a bucket for the user who posted the tweet. To retrieve a home timeline, we aggregate the 10 most recent tweets of the user's followees and sort them by timestamp in descending order and then select the first 10.
3. Redis Database Keys and Values

Keys	**Values**
tweet_<id> is the key for each tweet that is posted, id is a unique attribute (integer)	The tweet contents are stored as a serialized String (timestamp is stored as Unix Time in milliseconds: "tweet_id|user_id|timestamp|text" **Ex.** "17|972|294845392|this is a tweet"
timeline_<id> is the key for the user's timeline who has the corresponding id	The timeline is stored as a Redis list of tweet id's (integers) **Ex.** [1,17,9034,100]
tweets_<id> is the key for the user's personal tweets who had the corresponding id	The tweets are stored as a Redis list of tweet id's (integers) **Ex.** [1,17,9034,100]
users is the key for the set of unique users of Twitter	The users information is stored as a Redis set of user id's (integers) **Ex.** {1,2,10,99}
follows_<id> is the key for the list of followers of the user with the corresponding id	The list of followers is stored as a Redis list of user id's (integers) **Ex.** [3,89,100]
followees_<id> is the key for the list of followees of the user with the corresponding id	The list of followees is stored as a Redis list of user id's (integers) **Ex.** [3,89,100]

next_tweet_id is the key for the unique integer value that represents the id that the next posted tweet will be assigned	The id value is stored as an integer and is incremented each time after it is assigned to a posted tweet to maintain uniqueness **Ex.** 17

PERFORMANCE TESTING

Strategy 1:

1. Approximately **9571.94** tweets can be posted per second. Because Twitter receives 6-10 thousand new tweets per second, Redis can keep up.
2. Approximately **1010.10** home timelines can be retrieved per second. Our program is far from being within an order of magnitude of the 200-300 thousand home timeline refreshes that happen per second, so it is not close to keeping up.

Strategy 2:

1. Approximately **5235.93** tweets can be posted per second. Because Twitter receives 6-10 thousand new tweets per second, Redis can almost keep up.
2. Approximately **3357.48** home timelines can be retrieved per second. Our program is still not within an order of magnitude of the 200-300 thousand home timeline refreshes that happen per second, so it is not close to keeping up.

ANALYSIS AND REPORTING

Hardware configuration: 1.4 GHz Quad-Core Intel Core i5 processor
Software stack: Redis 6.2.6, Java (Jedis, java.sql, java.util, java.io, java.time)

API Method	**API Calls/Sec**
postTweet **(Strategy 1)**	9571.94
getHomeTimeline **(Strategy 1)**	1010.10
postTweet **(Strategy 2)**	5235.93
getHomeTimeline **(Strategy 2)**	3357.48

Factors that impacted our results:

- Concurrently running applications, particularly ones which require lots of RAM (IntelliJ, Youtube windows, etc.)
- Optimization: the particular backend data structure representation of a tweet can significantly impact post and retrieval performance (serialized Strings are faster than Hashsets)

- The particular strategy implemented also greatly impacts performance, as demonstrated by our performance test results. Strategy 1 was much better in terms of insertions, but its retrieval performance was drastically worse than that of Strategy 2