TEAM 6

Jinwoo Lee, Jongsoo Han, Kunal Jangam, Heon Park

Introduction

This project is about storing tweet data by leveraging our knowledge of the database that we learned during the class. The goal of this project is to store data efficiently so that users can get fast access to the data they want.

Dataset

The Twitter dataset is in JSON format. The number of tweets is over 90,000 and the data set has more than 20 fields. It is a quite large dataset so we have to make sure to store the data efficiently

Persisted Data Model and Datastores

We have decided on using a relational database in PostgreSQL to represent users and a non-relational database in MongoDB to represent tweets.

User data store design: PostgreSQL

	screen_name [PK] character varying (100)	user_id bigint	followers_count... integer	tweet_ids bigint[]
1	nuffsaidny	16144221	50	{1249378751349231616}
2	umesh_agr	268218622	24	{1249326224964345857}
3	meysimek	1193535233242664960	4	{1249403114614075400}
4	biannagolodryga	14135350	6343	{1249316363681910784}
5	ani_royal007	3917836273	74	{1248907321947783170}

First five row of our user database(PostgreSQL)

Our design for the user table includes columns for:

- Screen_name as primary key
- user_id and screen name for meaningful output

- followers count as a measure of rank
- list of tweets that the user tweeted or retweeted

In the case of a retweet, we will store the tweetid of the original id instead of the tweetid of the retweet. This is helpful when searching for all the tweets and retweets for a specific user, because of our tweet data storage design. As we set the primary key for screen_name this plays a role as an index in PostgreSQL. As the screen_name is the primary key, it has an advantage for searching by user. But when we want to find popular or famous users based on followers_count value, we have to order by the data to get top users. This is the disadvantage. To improve this weakness, the popular data is stored in Redis.

Tweet data store design: MongoDB

Example of our tweet database (MongoDB)

Tweet data store fields:

- id: int, primary key
- text: string
- text_sorted: list of strings
- created_at: datetime(ISO format)

- user_id: int, foreign key
- favorite_count: int
- retweet_count: int
- interactions: int
- hashtags: list of strings
- retweets: list of retweet objects:
    - Retweet object contains:
        - id: int, primary key
        - user_id: int, foreign key
        - created_at: datetime(ISO format)

The interactions field is the sum of the favorite count and retweet count. It is a measure of the popularity of the tweet and this field will be used to rank search output.

A set of helper functions were used to store the data.

- get_JSON_date: takes in the created_at time field from the data and transforms it into ISO format. This is done because MongoDB internally converts this data into BSON native data objects. This process allows created_at data to be compared, allowing for search by range of dates.
- get_text_sorted: takes a string text field. Changes text to lowercase, splits string into a list, removes duplicates from the list by casting it to a set then back to a list, removes punctuation for every word in the list, sorts the list so binary search can be used, and returns the list for storage in the text_sorted field.
- Get_hashtags_sorted: sorts the list of hashtags and returns the list.

Several indexing and performance tradeoff decisions were made to improve the response time of the search application.

- Index on id: improves internal search time when inserting data into the data store.
- Index on user_id: improves search time when searching by user.
- Index on created_at: improves search time when searching by time range.
- Sorted text_sorted and hashtag fields: able to run binary search on the sorted data.

While sorting the text field and hashtag field improved search time, it also added a new column to the data store and also increased time taken when inserting. However, since there is a tweet character limit, and since the application is search heavy rather than insert heavy, this downside could be overlooked. However, using MongoDB's special text search index drastically reduced search time for the text search, so the application will use this special index over using a sorted text column in the final design. Unfortunately, only one of these special indices can be created, and would not be able to be used for the hashtag field. A possible option is creating a hashtag table that mapped hashtags to tweets, however as a result a massive amount of duplicate data would have to be stored. Another option includes appending hashtags to the text data and storing the result in a new column and using the special index on that column, however doing so would mean creating a new column of duplicate data and having to sift through text data to get to hashtag data during hashtag search, and vice-versa for text search. The final design of the application uses binary search on the sorted hashtag field.

Processing tweets for storing in datastores

Storing User Data: pscycopg2

1. Connect to PostgreSQL locally.

2. Create user database and table in PostgreSQL
3. When iterating the dataset, check whether the current screen_name is in the user database or not. If not in the user db, then insert the user data that we are interested in.
4. If the screen_name(PK) is already in the database, then check whether the current tweet_id is in the database. If not in the database, then append the tweet_id to the corresponding user row in the tweet_ids field.

Storing Tweet Data: pymongo

1. Connect to MongoDB locally. Initialize tweet database and collection.
2. Iterate over all tweets in file.
3. Extract tweet id from object and query collection for matches. If matches are found, go to the next tweet. If none are found, it should be stored:
4. Check if the tweet is a retweet(text field starts with RT). If so, extract the id, user_id, and created_at values from the object. These will be added to the retweets field of the tweet this retweet is referencing:

a. Check to see if the original tweet exists. If it does, update the original tweets retweets list using the pymongo update_one function with the $push operation. If the original tweet does not exist, extract all relevant fields from the retweetd_status sub object and insert the tweet to the data store, and if needed, the user as well.

b. IF STATEMENT FROM STEP 4 IS FALSE: extract all relevant fields from the object and insert the tweet to the data store. Aforementioned helper functions used to format text_sorted, hashtags, and created_at fields.

Search Application Design

All output tweets are sorted by interaction values in descending order.

search_by_user(): Search by user methods uses two functions: one gets tweet Id by user name and then gets tweet by ids to find tweets related to the input user names. First get tweet id by user name uses “select tweet ids from user db”

comparing the input screen name Postgres query to find tweet ids for the input user name And get tweet by ids simply finds the tweet in the input tweet IDS.

search_by_text(): input is the string that the user is interested in. Then this function returns the tweets that contain the input string. We leveraged the binary search logic mentioned above.

search_by_text(): input is the string that the user is interested in. Then this function returns the tweets that contain the input string. We used a string index in the MongoDB function.

search_by_ht():input is the hashtag that the user is interested in. Then this function returns the tweets that contain the input hashtag.

time_range_search(): By formatting the user input, we can set the start date and the end date to use in the tweet find query. For The query, we are using greater and less than query against the created at field.

retweet_info(tweet_id): The input is tweet id, and the output is the retweet information of the tweet related to the tweet id. such as user ID, screen name, and when it was retweeted at. To search for retweet info, we can find the input tweet using the tweet id, and then we can return the retweet info that we already stored in the database. So with the input of tweet id, we would get the output of the retweet information of the tweet related to the tweet id. and that are such as user ID, screen name, and when it was retweeted at.

other_tweet_by_author(author,tweet_id): The input is the author and the current tweet_id. This function returns the other tweets posted by the input author.

Caching Data with Redis

Redis is an open-source, in-memory data structured store, used as a database and cache. Since retrieving data from memory us a lot faster than retrieving data from disk, we used Redis to store popular data into cache.

For user data, the top 10 users are stored in Redis based on the number of followers. For tweet data, the popular tweets are stored in Redis based on the interactions (=favorite_count+retweet count) value that we devised.

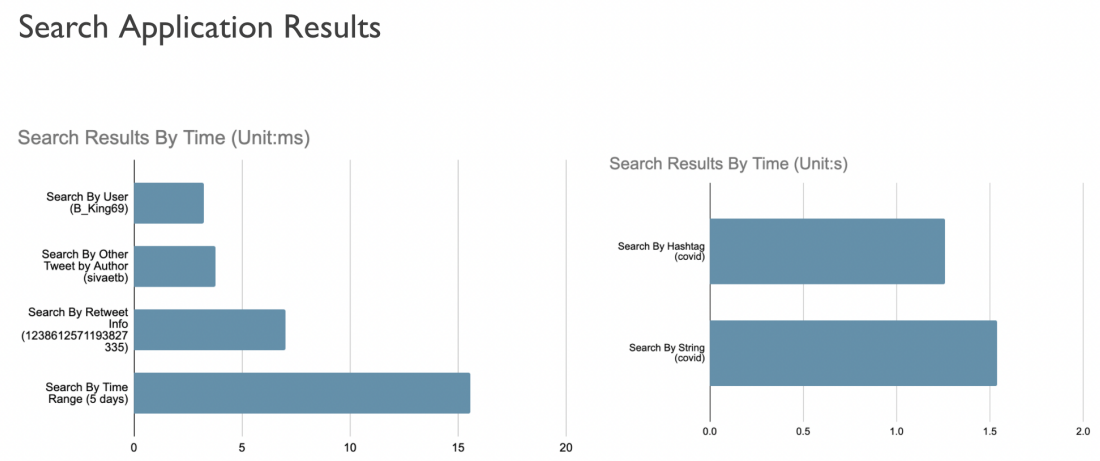
We used three main functions: store, get and update, to store our data in Redis from our databases, and also to return the data we stored in Redis.

A set of Redis functions were used to store the data.

- redis_store_top10_user(usertop10) : store the input data to our Redis DB. The input is in dictionary format which is key, value pair. And we used a sorted set as we are focusing on the ranking.
- redis_get_top10_user(): return top10 users from our Redis DB
- redis_update_top10_user(top10user): update the current data stored in our Redis DB so that our Redis DB always stores the most recent data. The input is the most recent data. If the input data and current Redis stored data are different, then update the data
- redis_store_top10_tweets(tweet10): store the input data to our Redis DB. The input is in dictionary format which is key, value pair. And we used a sorted set as we are focusing on the ranking.
- redis_get_top10_tweets(): return top10 tweets from our Redis DB
- redis_update_top10_tweets(top10tweets)

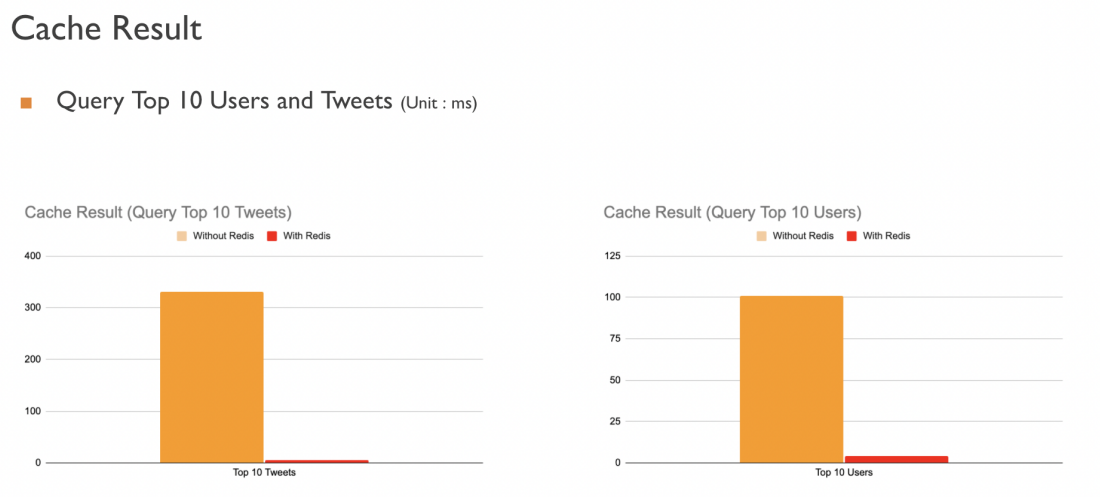
When we use cache, there is a stale data issue, in which the data in cache is not the most recent version committed to the database. Therefore, we used the "*redis_update_top10_user(top10user)*" and "*redis_update_top10_tweets(top10tweets)*" functions to compare the data in cache to the data in our databases, and store the latest version of the data.

Results



The result from our search applications

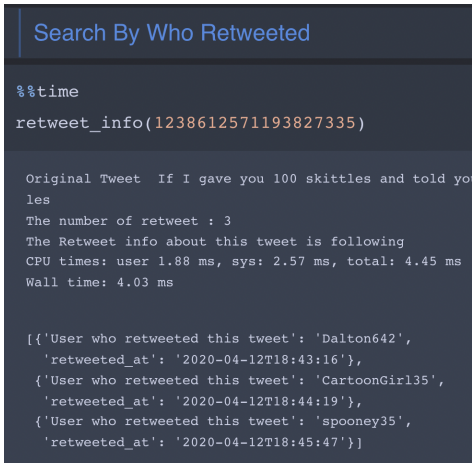
The left-hand side graph as you can see takes less than a second whereas the right-hand side search method takes little more than a second. This is due to us using the index for the left four search methods. The other thing interesting is that the time range consumed more time than the other searches in the left graphs. The reason why is that the other search functions used the index only one time but the time range used the time index twice. And that's why there is a time difference

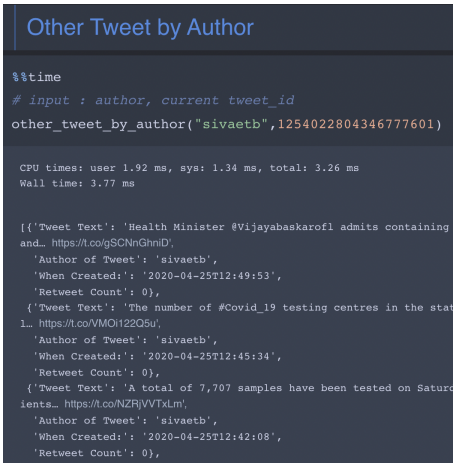


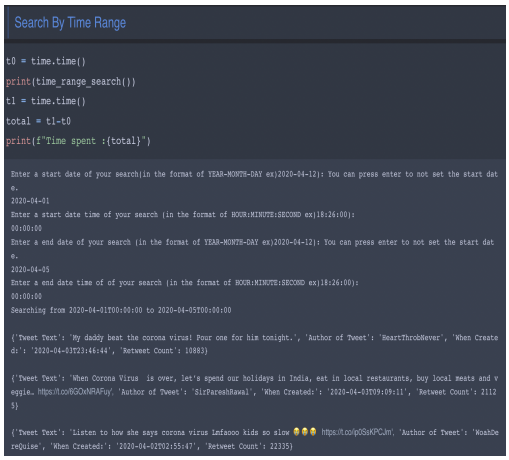
The result from comparing using Cache and not using Cache

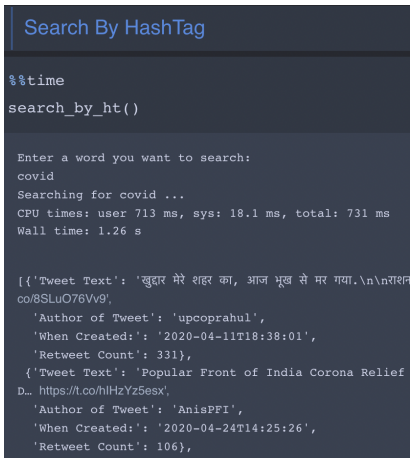
According to our results, Redis significantly helped reduce the search time compared to searching without Redes. This is because Redis retrieves the data from the memory, which is more efficient and faster than retrieving from the database.

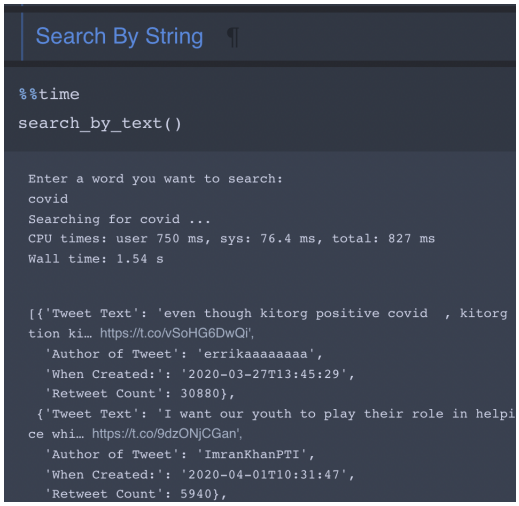
These are the actual results from our search applications.

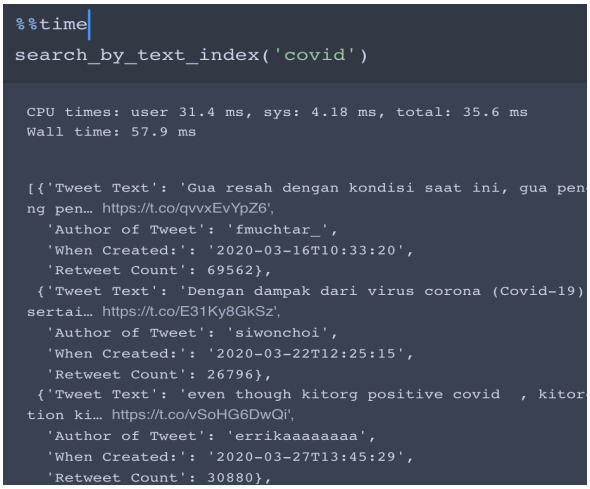












Conclusions

MongoDB's special text search index drastically reduces search time and is far better than the binary search implementation that was originally used. This is most likely due to the massive amount of tweets far exceeding the length of each tweet. As a result, it is much more efficient to optimize on the number of tweets through indexing rather than trying to optimize on the length of each tweet through querying every tweet and using binary search on the text field.

Redis is an in-memory data structured store, and is used for cache in our project. The popular data like top 10 users and tweets was successfully stored in Redis. Retrieving data from Redis was a lot faster than retrieving data from the database.

All in all, most importantly, through successfully completing the project, we are highly confident that we can deal with any kind of database for our future work in the industry.

References

https://www.postgresqltutorial.com/

https://www.postgresql.org/docs/current/tutorial.html

https://developer.twitter.com/en/docs/twitter-for-websites

https://www.mongodb.com/docs/guides/server/drivers/

https://www.w3schools.com/python/python_mongodb_getstarted.asp

https://redis.io/commands/zadd/

https://www.tutorialspoint.com/redis/sorted_sets_zadd.ht

Contributions

- Jinwoo Lee: relational data store and test the code
- Kunal Jangam: non relational data store
- Jongsoo Han: cache
- Heon Park: search application