Rahul Malhotra
Yuhsiang Hong
Jiazhang Cai
Hang Qi

Twitter Search Application Final Report

Introduction

For our project, we have collected tweets which include the hashtag “coronavirus”. The beauty of twitter is that tweets can have hashtags within them. In terms of databases, these hashtags act as keys that we can look for and group data by. Since the coronavirus topic is very popular and trendy at the moment, we had no issue collecting a large number of tweets. Based on what we have seen so far in the media, there are lots of different topics people are discussing. The main one is the virus itself and all the health-related aspects of it, such as symptoms, precautions, etc. However, a lot of secondary discussion is going, such as what people are doing and how they are coping with being quarantined.

The current condition we are in is truly like no other and it has resulted in a rather interesting time. Obviously, the mean threat and issue is dealing with the virus and preventing its spread. However, we were fascinated by the large-scale effects of this virus and sought to gain a glimpse of the world’s sentiment through these tweets.

Data Collection

Jiazhang first signed up for a Twitter developer account and used the API key to access the Twitter data. After we got the API, we used Jupyter Notebook and package “tweepy” to scrape twitter data with key “#coronavirus”. We interrupted the scrapping process as the number of tweets exceeded 10,000, and ended up collecting a total of 19,171 tweets.

Data Storage

To store the data, we will use both relational and non-relational database systems. Our plan is to store the user information in a relational store and the tweets in a non-relational store.

A. Relational Database

Yuhsiang dealt with storing the user data in a relational database. There are many relational database management systems that we can use to store user information. The two we were deciding between were MySQL and PostgreSQL. PostgreSQL is an object-relational database, while MySQL is a purely relational database. Namely, PostgreSQL includes features like table inheritance and function overloading, which can be essential to certain applications. The other great feature of PostgreSQL is that it can have a very powerful and useful Python API, called “psycopg2”, which enables us to interact with both PostgreSQL and MongoDB databases through Jupyter Notebook without switching the platforms.

Initially, we analyzed the structure of twitter user’s data. In our data, each user contained either 38 or 39 keys because some users didn’t have “profile_banner_url”. Second, we kept the keys that would be used for querying such as user “id”, “name”, “followers_count” and so on, and then we dropped the rest which were not important to us such as “profile_background_color”. Therefore, there would be 11 keys remaining in each user data as we can see on the graph below.


```
{'id': 531629036,
 'name': 'Creeds Cannon',
 'screen_name': 'ThucydidesTried',
 'protected': False,
 'verified': False,
 'followers_count': 11697,
 'friends_count': 12307,
 'listed_count': 72,
 'favourites_count': 73023,
 'statuses_count': 107291,
 'created_at': 'Tue Mar 20 20:53:16 +0000 2012'}
```

After that, we set the remaining 11 keys to be our columns and id as our primary key since each user's id was unique. Importing data was actually a tricky part because each user's data was an object with pairs of keys and values in JSON format. We could not directly import data as we did in MongoDB. Thus, we implemented "for" loops to iterate all the keys. During each loop, we used the "if" function to select the keys and created a temporary tuple to store their values. To load the tuple into the table, we used cursor.execute() and cursor.commit() functions. One important point was that we should set the PostgreSQL command "ON CONFLICT" for id, which enabled us to update the existing users' information. The graph below displays the users table from pgAdmin.

	id [PK] bigint	name text	screen_name text	protected boolean	verified boolean	followers_count bigint	friends_count integer	listed_count integer	favourites_count bigint	statuses_count bigint	created_at text
1	47868745023488	🔥 Sim...	Simargl4	false	false	3435	3540	0	4853	6505	Tue Jan 15 05:...
2	53338411245569	José	rjose316	false	false	14	68	0	2608	568	Fri Nov 08 17:5...
3	156549858	mceb72	MCEB72	false	false	5592	6087	12	19474	39468	Thu Jun 17 06:...
4	55374944	Debbie ...	debbiered15	false	false	3678	4969	4	97725	50267	Thu Jul 09 21:...
5	335037182	La Gue...	MayVenezolana	false	false	47420	2696	129	41370	192387	Thu Jul 14 01:...
6	72009411772416	Shark R...	SharkRadioNet	false	false	1072	2	14	276	57159	Mon Feb 10 20...
7	173211118	Healthy...	HealthAdv0cates	false	false	570	362	0	213	7757	Sat Jul 31 19:0...
8	58597033492484	Julio M...	ManonellasJulio	false	false	225	112	1	27600	9219	Sat Apr 08 03:5...
9	474029401	melelani	melelani22	false	false	185	73	8	75951	10394	Wed Jan 25 15...
10	15831179	Stay H...	YuriArtibise	false	false	9723	8304	676	98632	31811	Wed Aug 13 01...

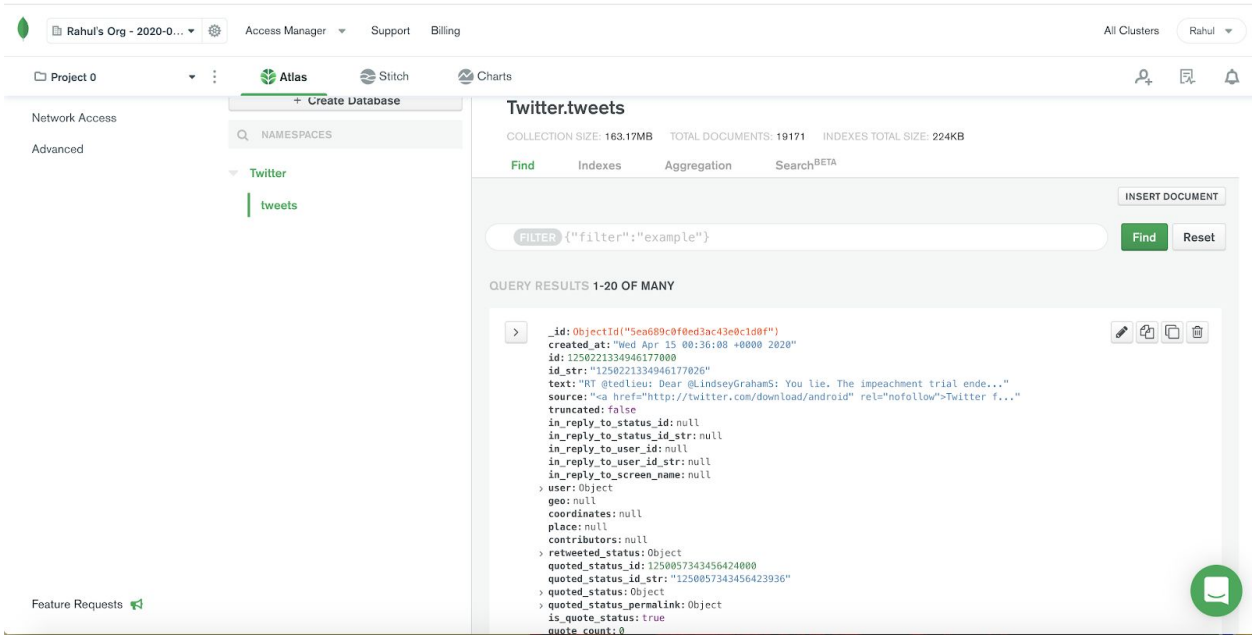
B. Non-relational Database

Rahul, Jiazhang and Hang dealt with storing the tweets data into a non-relational database. For the tweets data, we decided to use MongoDB to store it. One advantage of using MongoDB is that it is specialized for inserting certain types of data which are returned by social media platforms. The tweets were difficult to put into any table because the data is basically a JSON file with many objects which may have very different structures from other tweets, so we would have had trouble designing tables containing all the information from tweets data. However, we didn't have to worry about it in NoSQL databases such as MongoDB. Since MongoDB stores data in BSON files which are JSON style files, all we needed to do was collect the tweets information from Twitter and ask MongoDB to insert it in our database. MongoDB

simplifies the storing, searching, and recalling tweets. Our plan to store the tweets data was follows:

1. Install and run MongoDB service on our computers. We'll work with a MongoDB database from Python using PyMongo because the structure of JSON files is similar to dictionaries in Python.
2. Establish a connection to MongoDB. We are going to use the MongoClient object and get the database and collection that we are going to work with. If the database or collection does not exist, MongoDB will create it for us.
3. Create an index on the collection (table) to avoid the insertion of duplicate tweets. Namely, there will not be two tweets with the same index (id_str) inserted into our database.

In addition to storing the data using PyMongo, Rahul was able to sign up for an account on Atlas, which is MongoDB's cloud service. This was done by connecting to the cluster through the terminal and then using "mongoimport" to import the data into the primary shard. The benefit of importing data into the cloud was that it offered a convenient way to view our data, especially since it contains nested JSON objects. We were able to pick certain objects we wanted to look at, such as "user" and "retweeted_status" one at a time which made it easy to see what they contained. In addition, uploading the data to the cloud allows for collaboration and easy access.



Search Application

Indexing

1. PostgreSQL

We created two indexes in our users table in PostgreSQL database. The first index was for numbers of followers (followers_count). The second one was a multicolumn index for verified and followers_count. The purpose of these is that we want to find the users with certain numbers of followers or verified, so indexing on these enables us to query the result with shorter running time. The default method in PostgreSQL is BTree which is suitable for most of the situations. The graphs below display the execution times for finding the verified user with least numbers of followers before and after indexing.

```
query_data("EXPLAIN ANALYZE SELECT * FROM users WHERE verified = true ORDER BY followers_count ASC LIMIT 1", "analyze")
('Limit  (cost=478.76..478.76 rows=1 width=100) (actual time=2.404..2.404 rows=1 loops=1)',)
('  ->  Sort  (cost=478.76..479.41 rows=261 width=100) (actual time=2.404..2.404 rows=1 loops=1)',)
('        Sort Key: followers_count',)
('        Sort Method: top-N heapsort  Memory: 25kB',)
('        ->  Seq Scan on users  (cost=0.00..477.45 rows=261 width=100) (actual time=0.009..2.340 rows=261 loops=
1)',)
('              Filter: verified',)
('              Rows Removed by Filter: 15784',)
('Planning Time: 0.497 ms',)
('Execution Time: 2.436 ms',)
```

```
query_data("EXPLAIN ANALYZE SELECT * FROM users WHERE verified = true ORDER BY followers_count ASC LIMIT 1", "analyze")
('Limit  (cost=0.29..6.49 rows=1 width=100) (actual time=0.497..0.497 rows=1 loops=1)',)
('  ->  Index Scan using followers_verified_idx on users  (cost=0.29..1619.16 rows=261 width=100) (actual time=0.49
7..0.497 rows=1 loops=1)',)
('        Index Cond: (verified = true)',)
('Planning Time: 0.589 ms',)
('Execution Time: 0.531 ms',)
```

As we can see, the execution time dropped from 2.436 ms to 0.531 ms after indexing.

2. MongoDB

We created several indexes in our MongoDB database. If not creating indexes, it may require traversal the whole table to find the information. With index, (MongoDB stores indexes

using BTree data structure), the search time complexity can be reduced a lot. (Generally, it can reduce from O(N) to O(logN)). We created 4 single filed indexes.

The first one is by the number of followers under the “user” object.

The second one is by the twitter created time.

The third on is by the number of retweets under retweeted_status object

The fourth one is by the number of replies.

```
# create a single filed index for users' followers_count to speed up the access for each single tweet based
# on the number of their followers
collection.create_index([("user.followers_count", pymongo.DESCENDING)])

# create an index on "created_at"
collection.create_index([("created_at", pymongo.DESCENDING)])

# create an index on "retweeted_status.retweet_count"
collection.create_index([("retweeted_status.retweet_count", pymongo.DESCENDING)])

# create an index on "retweeted_status.reply_count"
collection.create_index([("retweeted_status.reply_count", pymongo.DESCENDING)])

# look at executionTimeMillis for search time, the smaller executionTimeMillis is better
# cursor.explain()
```

Querying

We did some sample queries in order to further support the search application. We first count the total number of tweets in the database, which is 19171. This picture is the output of the most recent tweet time, since this database was scraped 2 weeks before, the most recent date in the database is April 15th.

```
# find the time of the latest tweet created in this database
newest_tweet = db.first_100_tweets_table.find().limit(1)
for doc in newest_tweet:
    created_time_of_newest_tweet = doc['created_at']
print(created_time_of_newest_tweet)
```

```
Wed Apr 15 00:56:34 +0000 2020
```

We also have the output of the most recent twitter; user id with the largest number of followers, and the number of followers is more than 10 million. We have 167 users with more than 10 thousand followers. After this step, we were able to get answers to some common questions.

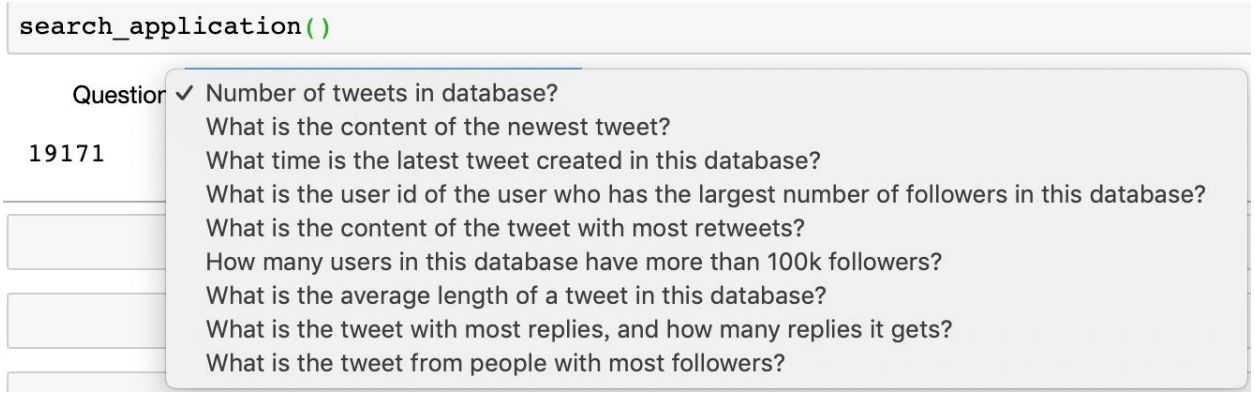
```
# find how many users in this database have more than 100k followers
cursor = db.twitter_collection.find({"user.followers_count": {"$gt": 100000}}, {"user.id", "user.followers_count"})
num_of_users_with_gt100k_followers = 0
for doc in cursor:
    num_of_users_with_gt100k_followers += 1
print(num_of_users_with_gt100k_followers)
```

```
167
```

Caching and GUI Design

Our first idea to deal with cache is to create a new collection which is the subset of MongoDB database. However, after our discussion, we realized that it was not a good method because cache should be the memory level instead of disk. Then we got an idea which is to create a dictionary to store the answers for some common questions as cache, then we can query from cache to get answers very fast.

Then we designed a basic search application User Interface based on common questions and queries we made before. There is a drop-down menu in our UI, so we can select from many questions shown in the picture.



As you can see in the picture, the cost of time between querying from cache and database is different: when we query the total number of tweets, we only need 4.86e-5 second but need 3.36 second to get the answer; similarly when we query the content of most recent tweet, it costs less time to query from cache rather than Database.

```
def common_question_in_cache():
    print(common_questions["Number of tweets in database?"])

import timeit
print(timeit.timeit(common_question_in_cache, number=1))
```

```
19171
4.860399999984111e-05
```

```
def common_question_outof_cache():
    cursor = db.twitter_collection.find()
    num_of_tweets = 0
    for doc in cursor:
        num_of_tweets += 1
    print(num_of_tweets)
import timeit
print(timeit.timeit(common_question_outof_cache, number=1))
```

```
19171
3.3622609230001217
```

Optimization

New UI

We optimized our search application in 2 ways. The first way is to design a new user interface to search more types of questions including search for the most recent tweets, famous users, frequent words, and tweets from famous users. In this picture, there is also a drop-down menu to select the type of questions, and we can further select numbers by dragging the bar from left to right to select top 1 to top 500 for each type of question. For example, if you want to query the top 10 newest tweets, you can select the first option in the drop-down menu and drag the bar from 1 to 10.

```
serach_appliaction = SearchApplicationOne(10)
serach_appliaction.user_interface()
```

Question: Find Newest Tweets

Top: 1

```
No. 1
Wed Apr 15 00:56:34 +0000 2020 User ID: 22091137 User Name: Basu Ghosh Das
RT @TarekFatah: Pakistanis in Karachi defying orders not to congregate in mosques by creating makeshift mosques on
rooftops. Working hard t…

{('Find Newest Tweets', 1): <__main__.LinkedNode object at 0x1097ca3c8>}
```

LRU Cache

The second optimization is that we made a second cache using "LRU Cache", it is a cache replacement algorithm that removes the least recently used data in order to make room for new data. So, the cache will self-update every time we do the query. Here we created LinkedList to save cache data and I set the maximum as 10 (you can see in the picture above that the parameter in my UI is 10), it means if we do the different queries 11 times, the data from the first query will be deleted from cache. After we first run this chunk of code, as shown in this picture, there is only one data in the cache.

```
{('Find Newest Tweets', 1): <__main__.LinkedNode object at 0x109816a58>}
```

In the next picture, after I queried 11 times, the cache now contains 10 data from the second query to the 11th query.

```
{('Find Newest Tweets', 2): <__main__.LinkedNode object at 0x1097ca3c8>, ('Find Newest Tweets', 3): <__main__.Linke
dNode object at 0x1097a2be0>, ('Find Newest Tweets', 4): <__main__.LinkedNode object at 0x10979e390>, ('Find Newest
Tweets', 5): <__main__.LinkedNode object at 0x10956bda0>, ('Find Newest Tweets', 6): <__main__.LinkedNode object at
0x1095f0ba8>, ('Find Newest Tweets', 7): <__main__.LinkedNode object at 0x1094836a0>, ('Find Newest Tweets', 8): <_
_main__.LinkedNode object at 0x1097ad400>, ('Find Newest Tweets', 9): <__main__.LinkedNode object at 0x1097c32e8>,
('Find Newest Tweets', 10): <__main__.LinkedNode object at 0x10987cba8>, ('Find Newest Tweets', 11): <__main__.Link
edNode object at 0x10980ffd0>}
```

Division of Project

Yuhsiang: Set up PostgreSQL database; Imported data into PostgreSQL; Created indexes for certain columns in users table; Implemented querying functions on users table and presented the results on Jupyter Notebook. Analyzed execution times of querying before and after indexing.
Jiazhang: Applied for twitter API; Data scraped; Imported data into MongoDB; Made Index for MongoDB data; Made sample queries for MongoDB data.
Hang: Imported data into MongoDB; Made sample queries for MongoDB data; Created 2 types of UI for search application; Created 2 types of cache by using dictionary(to store common questions and answers) and linkedNode(for LRU cache).
Rahul: Implementation of tweets data into MongoDB through the cloud; Made sample queries for tweet data; Helped come up with possible ideas for caching; Created indexes to speed up certain queries

Challenges

Throughout the process of building this search application, we came across several challenges. The first small challenge we faced was determining which databases to use to store our data. Luckily, we knew that we had to store the tweets data in a non-relational store and the user data in a relational store so this helped narrow down our options. Since we were able to collect the data as JSON objects, this motivated us to choose PostgreSQL and MongoDB.

Another major challenge we had to deal with was the caching system. First, we had to decide what entries we wanted to cache. Since the data we collected deals with the coronavirus, which is evolving on a daily basis, we wanted to try and store more recent tweets as users will want to keep up with news. Initially, we tried storing these newer tweets into a new collection within the database, but it did not give us any improvements in query time. Upon further research, we came across the LRU cache design and stored the tweets into a dictionary, where they would be accessed much more quickly from.

Scaling (Extra Credit)

The dataset we currently have is statis, that is, no new data has been added to it. However, we could make our data dynamic by continuously collecting and saving tweets. To do this, we would constantly need to be requesting tweets from Twitter (which could be cause for concern due to excessive usage from a single API key) and writing them into our file. Then, as they are being written into the file, we can save each tweet into our collection. Again, this would be extremely useful to our search application since it deals with news and adding data dynamically will keep the most recent tweets in our dataset. However, with this, we need to consider how to scale our database since it will grow relatively large. Luckily, MongoDB does a good job at handling large sets of data. Just on a single server, it can scale vertically to store up to 250 GB of data. If our dataset were to eventually exceed that amount, we would then need to make use of MongoDB's horizontal scaling, and distribute our data across several servers, which would likely need to be purchased.

Another aspect of a large database we would need to consider is making sure that our data is backed up. Fortunately, MongoDB's cloud version makes use of continuous backups to back up the data as it is being changed. Additionally, users can manually create a snapshot backup for which they can return to.

As the database scales with the size of the data, it is important that we also scale the cache as well. With more tweets in our database, there will be more tweets that we will want to have in our cache for quick access. One way to do this is to implement Redis into our database, which will create the key-value stores that we will save into our cache. Using Redis would allow us to increase the size of our cache and make it even faster in terms of performance.

Possible Improvements

While our search application is able to perform queries on the dataset we have acquired, we were able to think of several ways of improving it:

1. Modifying cache - currently we are storing the most recent tweets, however, one improvement we could make is to store the most recent tweets that meet a certain metric. Some of the metrics that a tweet could possess to qualify into the cache could be: being tweeted by a verified user, being tweeted by a user with a certain number of followers, or the tweet getting a certain number of replies, likes, and/or retweets. This would bring the tweets that are most likely to be accessed, making the most out of the efficiency of the cache.
2. Adding a language filter - currently our application does not distinguish between the language of tweets. When reading each tweet to add it into our database, we can add a "language" key for which the value is the language the tweet is written in. Python has several packages which help with language detection, such as "langdetect", "textblob", and "langrid", which we could use to assist us with this. This will be a nice addition since most people are only looking for tweets in one or two languages.

Conclusion

All in all, this experience was demanding, but rewarding. We were able to put into practice many of the concepts we learned in class and got to actually obtain and work with real data that is created and used everyday. We also got to see how some of the many databases we learned about are implemented into applications. Putting everything together, we came up with our search application and reflected on where and how it could be improved in the future.