Group 8
Kanya Kreprasertkul, Hui Wang, Varsha Rajasekar and Yelin Shin

Final Project Interim Report

Our project goal is to build up the databases and a search application from Twitter data using its API key. We choose “Corona” as our topic because in the current situation, it is a very popular topic throughout the world. Since we have not received the API key from Twitter, we used the sample dataset from Professor John to build up the databases. We used MongoDB to store tweet information, and PostgreSQL to store corresponding user information.

When we see the raw data, each object contains too much information with nested objects. Therefore, we set up the rules for which tweet and user we should save in certain cases. First, if the object is retweeting the other’s tweet, we only save the original tweet/user information into the databases. Since it does not contain extra comments like quoting a tweet, it is redundant to show when the user searches for the keyword from the tweet's content. Second, if the object is quoting the other’s tweet, we save only the quoting tweet, not the original tweet. However, we concatenate the extra comment and the original tweet’s text (format: extra comment || original content) because sometimes users do not use “Corona” in their comments, but the original content contains “Corona”. Otherwise, we save the original tweet’s information.

To build up a MongoDB database, we connect MongoDB and Jupyter notebook by PyMongo. Then, we create our database called “mydatabase” and the collection called “tweet”.

```
#making MongoDB database and collection
client = MongoClient("localhost", 27017)
db = client['mydatabase']
tweet = db['tweet']
```

Before we import the data into our MongoDB database, we pre-process the json object that contains only bottom information about Tweets.

Tweet Table:

id	id of tweet
user_id	id of the user
created_at	Time the tweet is generated
text	Tweet content
retweet_count	The number of retweets on tweet
favorite_count	The number of likes/favorites on tweet

reply_count	the number of replies on tweet
hashtags	list of hashtags

To follow our 3 rules, we check if an object contains a certain attribute or not. Therefore, we build a function called check_var that tries to run data[x] and if x does not exist in data, we return false otherwise return true. Also, this method is necessary to detect whether an object contains extended_tweet. From the raw data, we notice that if the original tweet's content is too long, twitter replaces the end of text by "...". Therefore, to save full text, we should check whether an object contains extended_tweet attribute. If it does, we collect data[extended_tweet][full_text] as our text.

```
def check_var(data, x):
    try:
        data[x]
        return True
    except:
        return False
```

Finally, we can put the pre-processed json object into our database by 3 conditions. We had a hard time collecting the correct data from the raw object because some objects contain a nested-nested object. For example, if someone retweets a quoted tweet, the object contains retweet_status and within retweet_status, there is quoted_status. Therefore, we use several if-else conditions in our code.

```
#case 1: check whether it is a retweet data or not
if (check_var(data, 'retweeted_status')) :

    Created_at = pd.to_datetime(data['retweeted_status']['created_at']).strftime('%B %d, %Y, %r')
    UserID = data['retweeted_status']['user']['id']
    ID = data['retweeted_status']['id']
    retweet_count = data['retweeted_status']['retweet_count']
    favorite_count = data['retweeted_status']['favorite_count']
    reply_count = data['retweeted_status']['reply_count']

    #check whether there is extension version of that tweet or not
    if (check_var(data['retweeted_status'], 'extended_tweet')) :
        Text = data['retweeted_status']['extended_tweet']['full_text']
        for t in data['retweeted_status']['extended_tweet']['entities']['hashtags'] :
            hashtag.append(t['text'])
    else :
        Text = data['retweeted_status']['text']
        for t in data['retweeted_status']['entities']['hashtags'] :
            hashtag.append(t['text'])

    #check whether it is retweet of quoted tweet or not
    if (check_var(data['retweeted_status'], 'quoted_status')) :

        #check whether there is extension version of the original tweet or not
        if (check_var(data['retweeted_status']['quoted_status'], 'extended_tweet')) :
            Text = Text + ' || ' + data['retweeted_status']['quoted_status']['extended_tweet']['full_text']
            for t in data['retweeted_status']['quoted_status']['extended_tweet']['entities']['hashtags'] :
                hashtag.append(t['text'])
        else :
            Text = Text + ' || ' + data['retweeted_status']['quoted_status']['text']
            for t in data['retweeted_status']['quoted_status']['entities']['hashtags'] :
                hashtag.append(t['text'])
```

(Sample code for retweeting condition, but we finished all of the cases)

In addition, if we get the same id while we import the Twitter's collection to our database, we update the retweet_count, favorite_count and reply_count because the numbers can increase by new retweeting data.

We have stored the user information in PostgreSQL database called twitter_users. In order to administer our database in a more friendly way, we used pgadmin that allowed us to create the database. PostgreSQL has a great performance handling JSON and has a very powerful python API, called psycopg2. We connected to our database on jupyter notebook using psycopg2.connect command and created a table called users_data containing the following attributes with user_id as our primary key.

User Table:

user_id	id of the user who tweeted
name	Name of the user
user_name	Twitter handle of the user
verified_status	User is verified or not
followers_count	Number of people following the user
friends_count	Number of people user is following
statuses_count	Number of tweets user has made
user_location	Location of the user
favourites_count	Number of tweets user has liked

Then, we defined a function store_data() that stores the required data in the user table. This function follows the same logic that we used to create the table - connect to the database, create a cursor, execute the query, commit query, close connection, but instead, we will use the INSERT INTO command. We might encounter multiple tweets of the same user. Since user_id is our primary key and primary keys must be unique it will raise an error as it detects that the user id is already in the table. So, we used the ON CONFLICT command here to tell postgreSQL it does not have to insert the user again. Every time the particular user is detected for insertion into the user table, it may have different values of some columns for example followers_count, verified_status, etc. Since the user data is dynamic, it is important to update the table simultaneously. With the ON CONFLICT command, we used DO UPDATE and declared the columns that may need update.

```
#This function takes the the variable inputs and stores it into a PostgreSQL database
def store_data(user_id, name, user_name, verified_status, followers_count, friends_count,statuses_count,user_location,f
    conn = psycopg2.connect("dbname=twitter_users port=5432 user=postgres password=password")
    cur = conn.cursor()
    insert_query = '''
    INSERT INTO users_data (user_id, name, user_name, verified_status, followers_count, friends_count,
    statuses_count,user_location,favourites_count) VALUES (%s, %s, %s, %s,%s,%s,%s,%s,%s)
    ON CONFLICT (user_id) DO UPDATE SET verified_status=EXCLUDED.verified_status,
    followers_count=EXCLUDED.followers_count, friends_count= EXCLUDED.friends_count,
    statuses_count=EXCLUDED.statuses_count, user_location=EXCLUDED.user_location,
    favourites_count=EXCLUDED.favourites_count
                '''
    cur.execute(insert_query, (user_id, name, user_name, verified_status, followers_count, friends_count,statuses_count
    conn.commit()
    cur.close()
    conn.close()
```

For storing the user information into the user table, we first grabbed the required variables from the raw json data. In the case of retweets, we want the data of the users who made the original tweets and not that of the users who retweeted. We implemented this case using the check_var() function which checks for the presence of retweeted_status in the objects. If the retweeted_status is not present in the object, then we grab the user information of the tweet in that object. Now that we stored the values in the respective variables, we passed them to the store_data() function that was defined earlier. A function create_pandas_table was then created that takes in the sql query and outputs the results in a pandas dataframe.

In [15]:
```
conn = psycopg2.connect("dbname=twitter_users port=5432 user=postgres password=password")
cur = conn.cursor()


# A function that takes in a PostgreSQL query and outputs a pandas database
def create_pandas_table(sql_query, database = conn):
    table = pd.read_sql_query(sql_query, database)
    return table
```

In [16]:
```
create_pandas_table("SELECT * FROM users_data ")
```

Out[16]:

	user_id	name	user_name	verified_status	followers_count	friends_count	statuses_count	user_location	favourites_count
0	303721121	Lal Saurabh Vikram S	lal_saurabh	False	174	366	4653	Newbury Berkshire UK	2680
1	288277167	Aaron Rupar	atrupar	True	303195	999	72439	None	100394
2	1229825409686081539	ez💥🔥👹@Evin Jiyanê👹✌️	maras_evin	False	3425	2928	4074	None	9764
3	2193495236	Lori Harvey	madeinIndia___	False	1084	689	19704	None	1172
4	1238805546230329347	Amersfoort Rainproof	033rainproof	False	70	423	58	Amersfoort, Nederland	48
...	...	...	...	...	...	...	...	...	...
9467	254407161	Mirasımız	mirasimiz	False	7816	52	6886	Fatih, İstanbul	275
9468	1139409272712773634	°•★•°~baby bun~°•★•°	Deepsdaniel	False	6	256	226	daniel playroom	945
9469	836123141243977732	Diskominfotik NTB	kominfotikntb	False	1160	84	1135	West Nusa Tenggara, Indonesia	243
9470	23831095	Dr. Atomreisfleisch	Atomreisfleisch	False	5497	976	224502	None	89643
9471	824215967198773248	#معا_ضد_كورونا	ghaith_alain	True	10004	0	1028	Ghaith@al-ain.net	44

9472 rows × 9 columns

With the tweet table and user table above, we want to design a search application which can satisfy all the needs of a user based on what information we have. We plan to use Spark doing the search in the tweet table instead of MangoDB, because Spark can use cache to save more time during the search. With the Tweet table, a user can get such as the most popular original tweet ranked by the retweets count; the

most popular hashtag ranked by aggregating the lists of hashtags among all tweets we collect. We plan to use PostgreSQL for searching in the user table. With the user table, a user can get such as the most active user ranked by tweets count; or the most active location ranked by tweets count. Of course, We also plan to join the two tables by user id to get some more comprehensive search.

When a user searches for a keyword or hashtag in the Tweet table, the results will show all the original tweets that contain it. We can rank it by number of retweets, number of favorites, number of replies, or sum of them to find the most popular tweet. We can also link the results with the user table to find out the corresponding users who tweet them and rank the results by the number of followers, number of friends, number of statuses, number of favorites or combination of them.

We also save the location information of users in the user table, even though it is self-reported and users can choose whether they want to register or not. We still plan to set the related search in our application. A user can search for the most preferred location ranked by amount of twitter users. When a user searches a hashtag, the results can show the most active location related to the hashtag.

Division of labour

- Kanya & Yelin: Data cleaning/importing for Tweet database
- Varsha: Data cleaning/importing for User database
- Hui: Designing the search application - later implementing.