Final Report

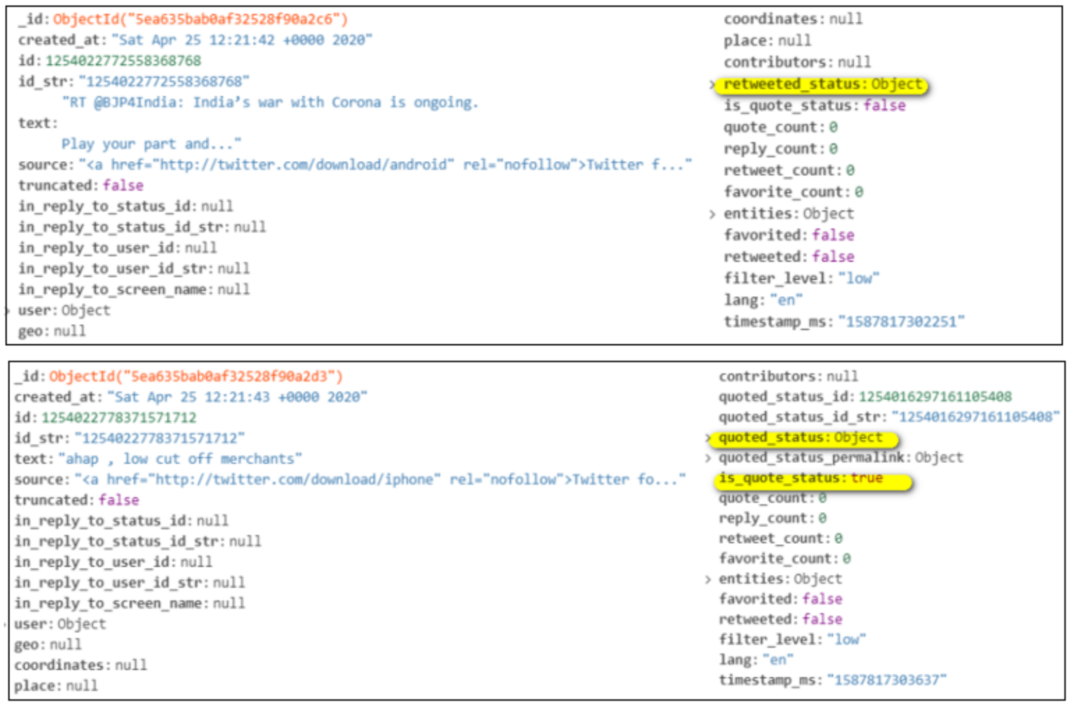
Group 6: Kanya Kreprasertkul, Hui Wang, Varsha Rajasekar and Yelin Shin

Twitter Search Application

Our project goal is to build up the databases and the 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 that contains 101K (101,916) objects from Professor John to build up the databases. We used MongoDB to store tweet information, and PostgreSQL to store corresponding user information. After that we used Pandas to merge two tables before printing out the search result.

Data Cleaning

Before cleaning the raw data, we looked into it to figure out what information we need to keep for displaying search results. While we investigated it, we noticed that some objects contain double nested objects when the user retweets a quoting tweet. Since the structure of raw data was too complicated to show every information, we set up the rules for which tweet and user we should save in certain cases shown in Figure 1.



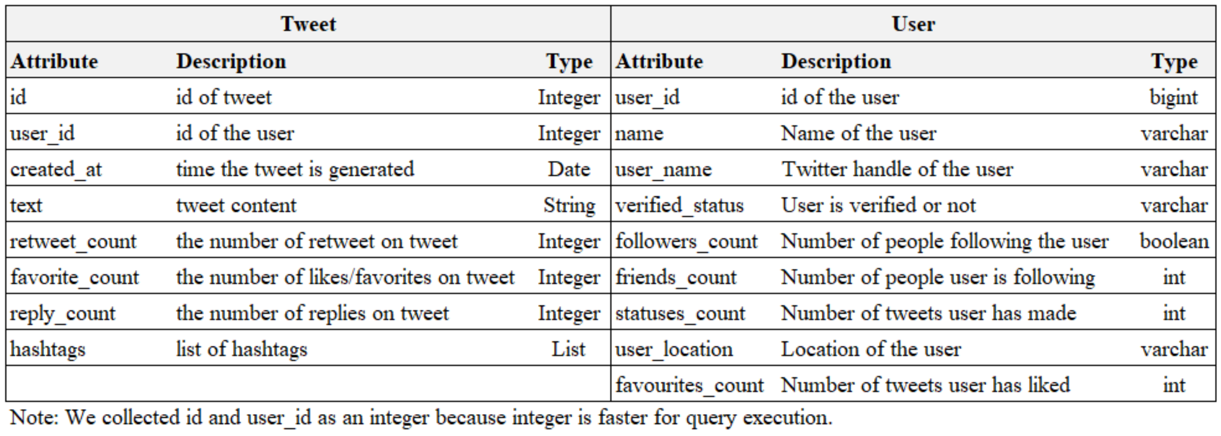
```
_id: ObjectId("5ea635bab0af32528f90a2c6")
created_at: "Sat Apr 25 12:21:42 +0000 2020"
id: 1254022772558368768
id_str: "1254022772558368768"
     "RT @BJP4India: India’s war with Corona is ongoing.
text:
     Play your part and..."
source: "<a href="http://twitter.com/download/android" rel="nofollow">Twitter f..."
truncated: false
in_reply_to_status_id: null
in_reply_to_status_id_str: null
in_reply_to_user_id: null
in_reply_to_user_id_str: null
in_reply_to_screen_name: null
user: Object
geo: null
coordinates: null
place: null
contributors: null
retweeted_status: Object
is_quote_status: false
quote_count: 0
reply_count: 0
retweet_count: 0
favorite_count: 0
entities: Object
favorited: false
retweeted: false
filter_level: "low"
lang: "en"
timestamp_ms: "1587817302251"
```

```
_id: ObjectId("5ea635bab0af32528f90a2d3")
created_at: "Sat Apr 25 12:21:43 +0000 2020"
id: 1254022778371571712
id_str: "1254022778371571712"
text: "ahap , low cut off merchants"
source: "<a href="http://twitter.com/download/iphone" rel="nofollow">Twitter fo..."
truncated: false
in_reply_to_status_id: null
in_reply_to_status_id_str: null
in_reply_to_user_id: null
in_reply_to_user_id_str: null
in_reply_to_screen_name: null
user: Object
geo: null
coordinates: null
place: null
contributors: null
quoted_status_id: 1254016297161105408
quoted_status_id_str: "1254016297161105408"
quoted_status: Object
quoted_status_permalink: Object
is_quote_status: true
quote_count: 0
reply_count: 0
retweet_count: 0
favorite_count: 0
entities: Object
favorited: false
retweeted: false
filter_level: "low"
lang: "en"
timestamp_ms: "1587817303637"
```

Note: In the status, it contains another tweet object that is being retweeted or quoted

Figure 1. Example of the nested object

First, if the object is retweeting the other's tweet, we only save the original tweet's 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's information, 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 regular tweet's information. Moreover, since we have 101K objects, we decided to store only English tweets. After we set up the rules, we opened the Twitter application to see what content they show in the search result. So, we settled the tables' content as shown below.



Tweet			User		
Attribute	**Description**	**Type**	**Attribute**	**Description**	**Type**
id	id of tweet	Integer	user_id	id of the user	bigint
user_id	id of the user	Integer	name	Name of the user	varchar
created_at	time the tweet is generated	Date	user_name	Twitter handle of the user	varchar
text	tweet content	String	verified_status	User is verified or not	varchar
retweet_count	the number of retweet on tweet	Integer	followers_count	Number of people following the user	boolean
favorite_count	the number of likes/favorites on tweet	Integer	friends_count	Number of people user is following	int
reply_count	the number of replies on tweet	Integer	statuses_count	Number of tweets user has made	int
hashtags	list of hashtags	List	user_location	Location of the user	varchar
			favourites_count	Number of tweets user has liked	int

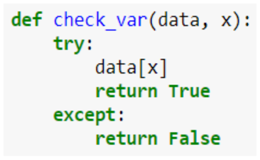
Note: We collected id and user_id as an integer because integer is faster for query execution.

Table 1. The tweet table's attribute description

Data Importing in MongoDB

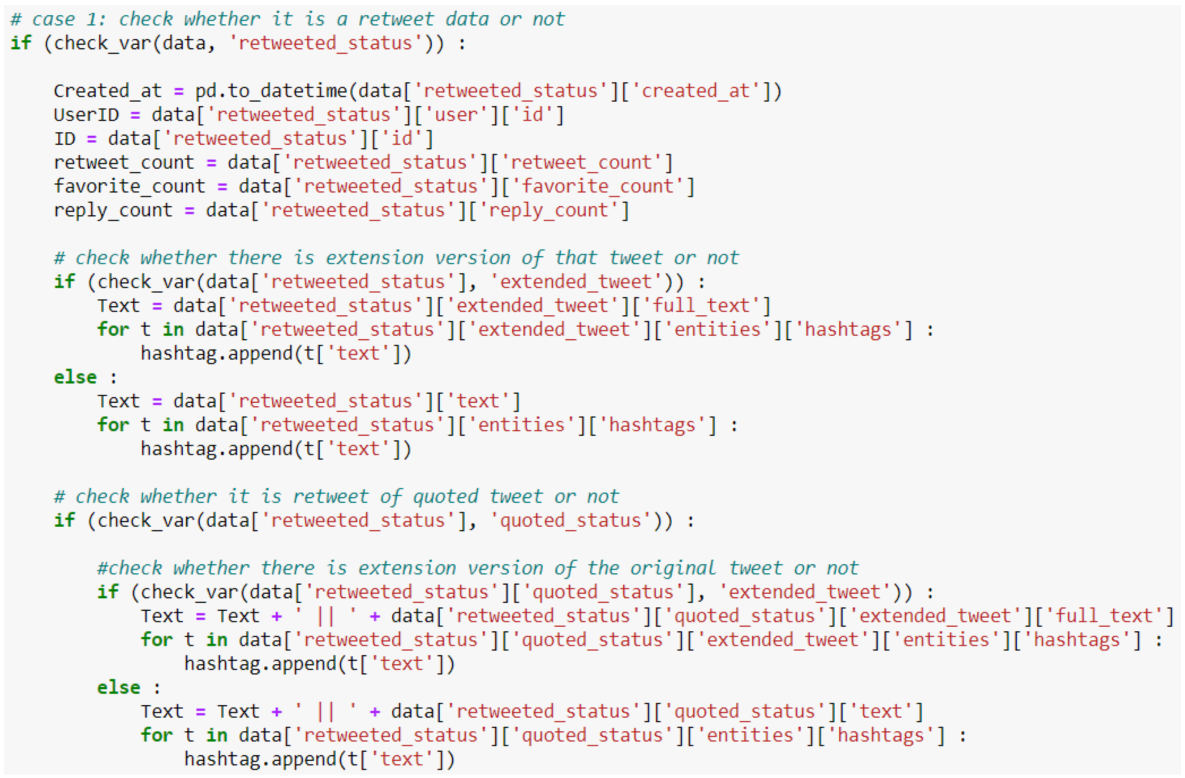
We used MongoDB which is a NoSQL database to store tweet information. The reason why we chose MongoDB is that it is very easy to install and setup. Being non-relational data structure makes it easier to handle our data because columns in our data are not quite consistent. It provides many powerful query processing. Besides, Indexes in MongoDB are easy to create and very helpful for queries or searches in the database. MongoDB also has many powerful graphical user interface tools, such as Studio 3T, MongoDB Compass and NoSQL Manager. We decided to use MongoDB Compass which is user friendly and can be installed directly from MongoDB website. We also researched MongoDB for our first project, so we think it is a good opportunity to use it in the final project too. Moreover, MongoDB can operate well with Python. We can easily connect MongoDB and Jupyter notebook using PyMongo.

After we connected MongoDB and Jupyter notebook by PyMongo, we created our database named "mydatabase" and the collection named "tweet". From the 3 rules that we mentioned earlier, we need to check whether an object contains a certain attribute or not, so we built a function named "check_var" which tries running data[x] as shown in Figure 2. If x exists in data, it returns True, otherwise it returns False. So, we used this function to check whether an object contains retweeted_status (whether the tweet is retweeted from the original tweet or not) and quoted_status (whether the tweet is quoted from the original tweet or not). We also used this function to check extended_tweet which contains the extended version of text as we found out that if the content of the tweet is too long, the end of that tweet will be shrinked and replaced with "...". Therefore, we need to go further into the nested attributes to save the full text.



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

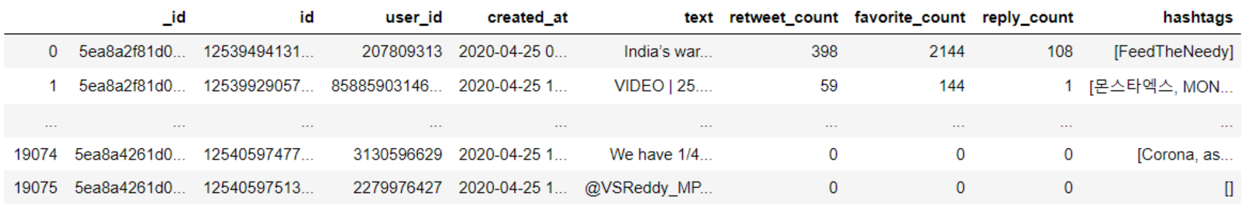
Figure 2. The check_var function



```python
# 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'])
    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'])
```

Figure 3. Example of our code to check the nested object

In addition, we encountered many nested-nested objects apart from the cases we mentioned above. For example, some users retweeted a quoted tweet, so that object contains retweet_status and within retweet_status, there is quoted_status. Also, we need to check extended_tweet within that nest. So, we built several if-else conditions to check the cases. Another case that we need to consider is that if we get the tweet with the same id that we insert in our database before (it means that they are the same tweet), we update the number that can increase, such as the retweet_count, favorite_count and reply_count. Also, to optimize our search application, we create indexes for tweet id, text, hashtags and created_at. Figure 4 is our Tweet table that we use Pandas to generate the table from our tweet collection.



	_id	id	user_id	created_at	text	retweet_count	favorite_count	reply_count	hashtags
0	5ea8a2f81d0...	12539494131...	207809313	2020-04-25 0...	India's war...	398	2144	108	[FeedTheNeedy]
1	5ea8a2f81d0...	12539929057...	85885903146...	2020-04-25 1...	VIDEO | 25....	59	144	1	[몬스타엑스, MON...
...	...	...	...	...	...	...	...	...	...
19074	5ea8a4261d0...	12540597477...	3130596629	2020-04-25 1...	We have 1/4...	0	0	0	[Corona, as...
19075	5ea8a4261d0...	12540597513...	2279976427	2020-04-25 1...	@VSReddy_MP...	0	0	0	[]

Figure 4. The Tweet table

Data Importing in PostgreSQL

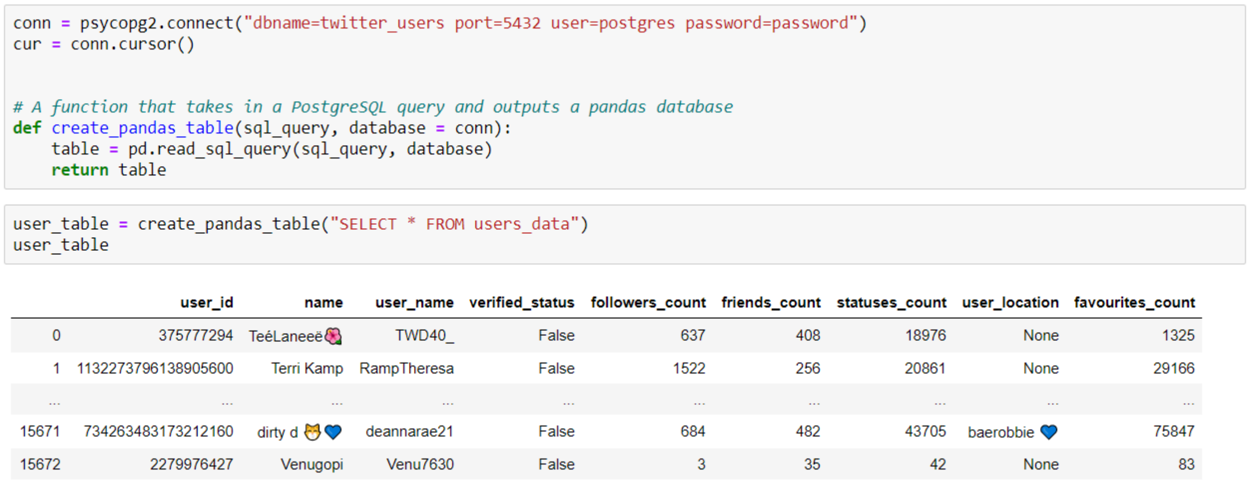
PostgreSQL, an open source object-r database, was chosen to store the user data. It uses both dynamic and static schemas and allows us to use it for relational data and normalized form storage. Data ingestion and data selection is faster in Postgres which also consumes less disk space compared to other SQL and NoSQL databases. Postgres has a great performance handling JSON and provides data constraint and validation functions, which help to ensure JSON documents are more meaningful. The big thing about Postgres is that it lets us keep our options open. We can either choose to route data to a JSON column, which can be modelled later, or we can put it into an SQL-schema table, all within the same Postgres database.

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. Postgres has a very powerful python API, called psycopg2. We connected to our database on the jupyter notebook using psycopg2.connect command and created a table called users_data with user_id as our primary key. PostgreSQL automatically creates an index for the primary key constraint; hence we did not have to create a separate index on user_id.

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 a 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 to be updated.

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 otherwise we grab the user information inside the user object of the retweeted_status. Now that we have acquired the values of the respective variables from the json file, 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.



```
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
```

```
user_table = create_pandas_table("SELECT * FROM users_data")
user_table
```

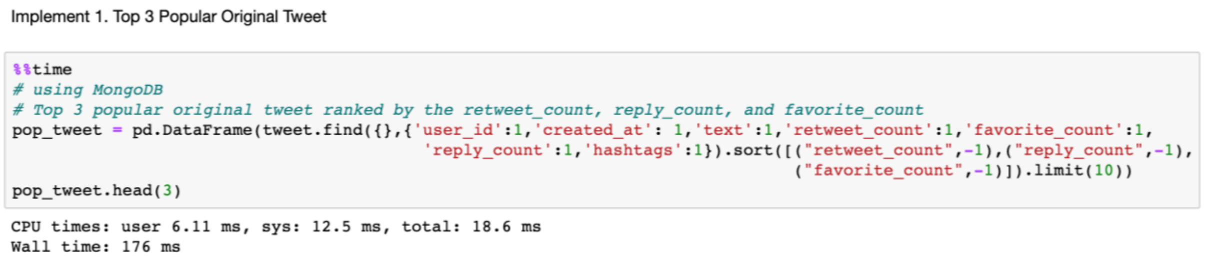
	user_id	name	user_name	verified_status	followers_count	friends_count	statuses_count	user_location	favourites_count
0	375777294	TeéLaneeë🌺	TWD40_	False	637	408	18976	None	1325
1	1132273796138905600	Terri Kamp	RampTheresa	False	1522	256	20861	None	29166
...	...	...	...	...	...	...	...	...	...
15671	734263483173212160	dirty d 😸💙	deannarae21	False	684	482	43705	baerobbie 💙	75847
15672	2279976427	Venugopi	Venu7630	False	3	35	42	None	83

Figure 5. The User table

Search Application Design

With the Tweet table and the User table we got previously, we want to design a search application which can satisfy all the needs of a user based on what information we have. We used MongoDB for the searching in the Tweet table, SQL for the searching in the User table, and Pandas to link the two tables and show the result. We considered the case insensitive, for example, if you search "corona" in lower letter, the one with capital letter will also come up in the result, which is the same as the searching case in twitter website. As you will see in the codes and results shown later, we recorded the running time for the searching implementations to check if our design is reasonable to save time and space as well.

Within the Tweet table, for implementation 1, a user can get the popular original tweet ranked by the retweet_count, reply_count, and favorite_count. To save running time and space, we only search 10 most popular original tweets in MongoDB using limit.() and save it to pandas. For implementation 2, a user can get the popular hashtag ranked by aggregating the lists of hashtags (case insensitive). We save the hashtags as a list in the database for every original tweet and count every element in the lists for ranking.

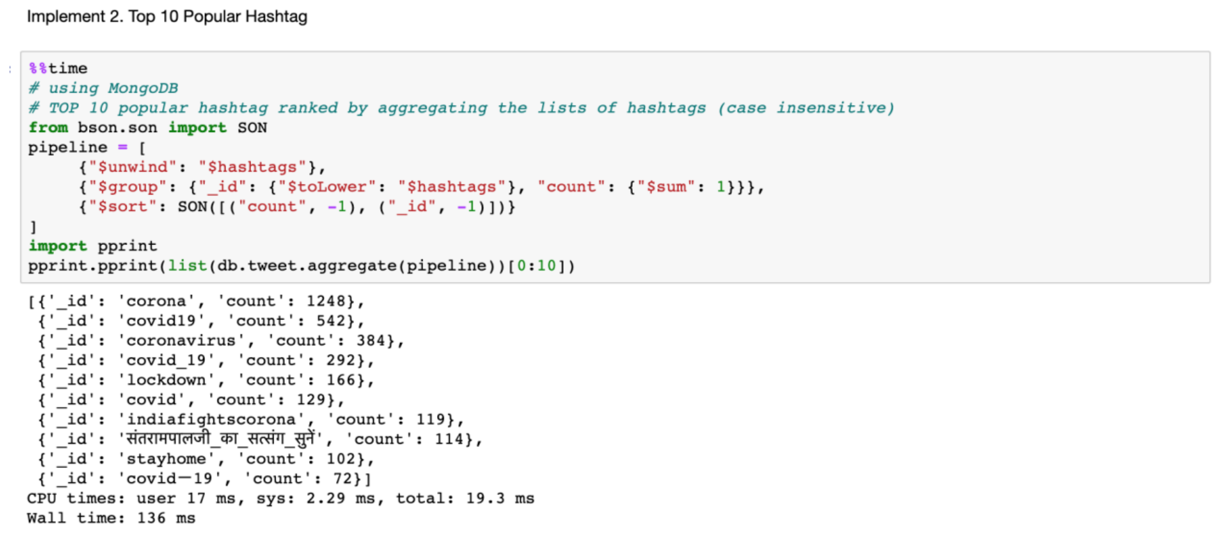


Implement 1. Top 3 Popular Original Tweet

```
%%time
# using MongoDB
# Top 3 popular original tweet ranked by the retweet_count, reply_count, and favorite_count
pop_tweet = pd.DataFrame(tweet.find({},{'user_id':1,'created_at': 1,'text':1,'retweet_count':1,'favorite_count':1,
                                        'reply_count':1,'hashtags':1}).sort([("retweet_count",-1),("reply_count",-1),
                                                                            ("favorite_count",-1)]).limit(10))
pop_tweet.head(3)
```

```
CPU times: user 6.11 ms, sys: 12.5 ms, total: 18.6 ms
Wall time: 176 ms
```

Figure 6. Implementation 1



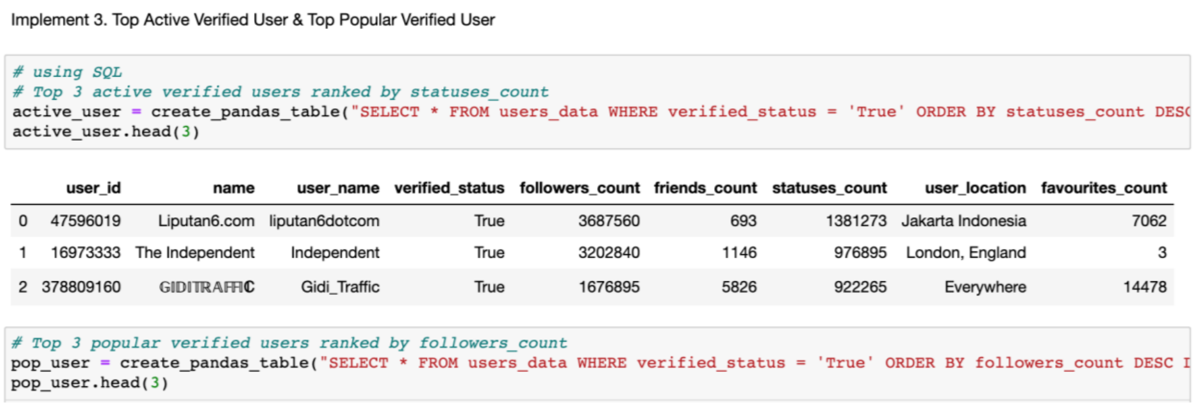
Implement 2. Top 10 Popular Hashtag

```
%%time
# using MongoDB
# TOP 10 popular hashtag ranked by aggregating the lists of hashtags (case insensitive)
from bson.son import SON
pipeline = [
     {"$unwind": "$hashtags"},
     {"$group": {"_id": {"$toLower": "$hashtags"}, "count": {"$sum": 1}}},
     {"$sort": SON([("count", -1), ("_id", -1)])}
]
import pprint
pprint.pprint(list(db.tweet.aggregate(pipeline))[0:10])
```

```
[{'_id': 'corona', 'count': 1248},
 {'_id': 'covid19', 'count': 542},
 {'_id': 'coronavirus', 'count': 384},
 {'_id': 'covid_19', 'count': 292},
 {'_id': 'lockdown', 'count': 166},
 {'_id': 'covid', 'count': 129},
 {'_id': 'indiafightscorona', 'count': 119},
 {'_id': 'संतरामपालजी_का_सत्संग_सुनें', 'count': 114},
 {'_id': 'stayhome', 'count': 102},
 {'_id': 'covid—19', 'count': 72}]
CPU times: user 17 ms, sys: 2.29 ms, total: 19.3 ms
Wall time: 136 ms
```

Figure 7. Implementation 2

For implementation 3, using SQL, a user can get the top active verified users ranked by statuses_count and top popular verified users ranked by followers_count. And, importantly, to save the running time and space, we also ranke 10 most active or popular users in SQL and save to Pandas dataframe, while the user can choose to show only top 3 or 5.


Implement 3. Top Active Verified User & Top Popular Verified User

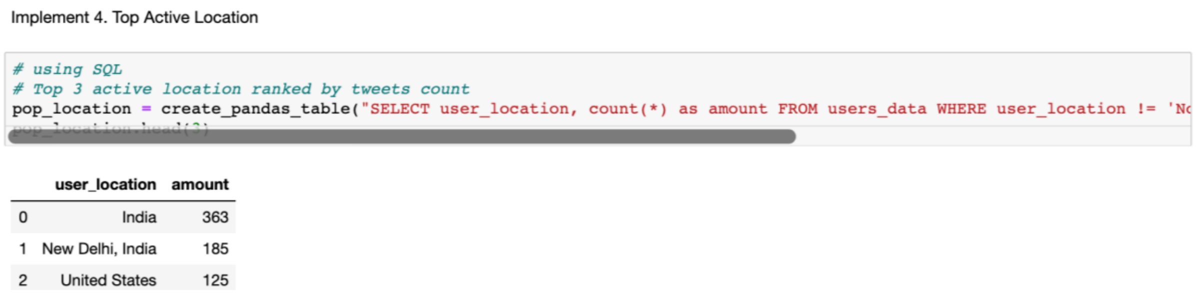
```
# using SQL
# Top 3 active verified users ranked by statuses_count
active_user = create_pandas_table("SELECT * FROM users_data WHERE verified_status = 'True' ORDER BY statuses_count DESC
active_user.head(3)
```

	user_id	name	user_name	verified_status	followers_count	friends_count	statuses_count	user_location	favourites_count
0	47596019	Liputan6.com	liputan6dotcom	True	3687560	693	1381273	Jakarta Indonesia	7062
1	16973333	The Independent	Independent	True	3202840	1146	976895	London, England	3
2	378809160	GIDITRAFFIC	Gidi_Traffic	True	1676895	5826	922265	Everywhere	14478

```
# Top 3 popular verified users ranked by followers_count
pop_user = create_pandas_table("SELECT * FROM users_data WHERE verified_status = 'True' ORDER BY followers_count DESC I
pop_user.head(3)
```

Figure 8. Implementation 3

Even though location information is self-reported, users in twitter can choose whether they want to register or not. We still choose to set the related search in our application, because we think during such a globalized era, locations contain lots of important information. So, for implementation 4, a user of our application can get the top active location ranked by counting the tweets posted in that location. And the result shows that twitter users in India prefer to show their location compared to others.

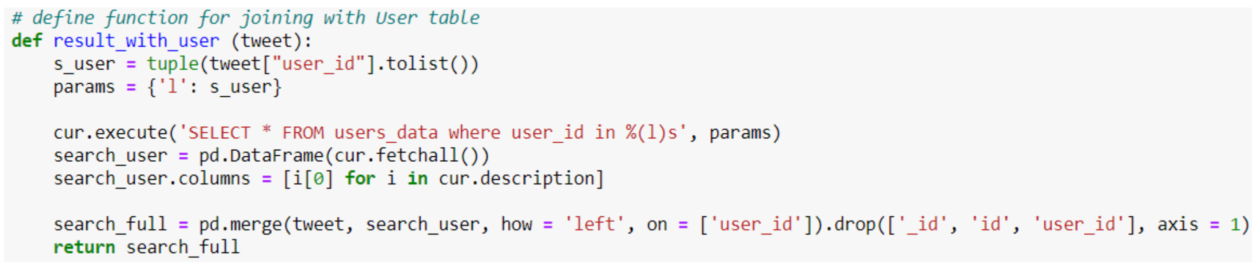

Implement 4. Top Active Location

```
# using SQL
# Top 3 active location ranked by tweets count
pop_location = create_pandas_table("SELECT user_location, count(*) as amount FROM users_data WHERE user_location != 'Nc
pop_location.head(3)
```

	user_location	amount
0	India	363
1	New Delhi, India	185
2	United States	125

Figure 9. Implementation 4

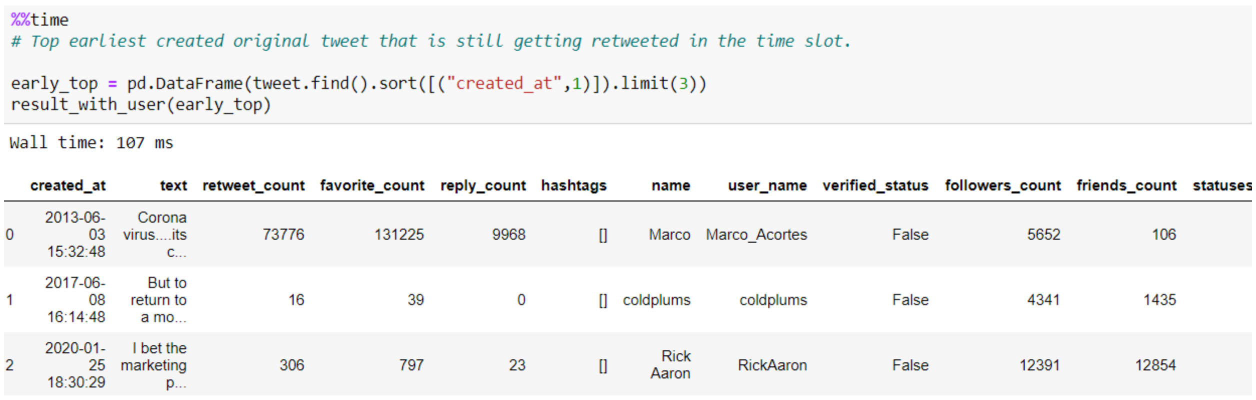
For further search applications, we create a function called "result_with_user" which uses Pandas to join tweets and users together to display the right format of search result as shown in Figure 10. It grabs the user_id from the tweet table and finds the corresponding user in the user table. Then we used pandas to join two data frames by user_id.



```
# define function for joining with User table
def result_with_user (tweet):
    s_user = tuple(tweet["user_id"].tolist())
    params = {'l': s_user}

    cur.execute('SELECT * FROM users_data where user_id in %(l)s', params)
    search_user = pd.DataFrame(cur.fetchall())
    search_user.columns = [i[0] for i in cur.description]

    search_full = pd.merge(tweet, search_user, how = 'left', on = ['user_id']).drop(['_id', 'id', 'user_id'], axis = 1)
    return search_full
```

Figure 10. The result_with_user function

Then, we searched the top earliest created original tweet that is still getting retweeted in the time slot in implementation 5 and we found a very interesting result. We know that the current pandemic, COVID-19, started at the end of 2019, but we found that a tweet created by the user named Marco in 2013 saying “Corona virus….its coming”. Is it a real tweet? If it is, then isn’t that very tricky? It seems like Marco can predict the future and knew COVID-19 would come years earlier than it did. After searching it on news, we think it might mean the Middle East respiratory syndrome (MERS). MERS, which also known as camel flu, is a viral respiratory infection caused by the MERS-coronavirus (MERS-CoV). The first identified case occurred in 2012 in Saudi Arabia and most cases have occurred in the Aranbian Peninsula.



```
%%time
# Top earliest created original tweet that is still getting retweeted in the time slot.

early_top = pd.DataFrame(tweet.find().sort([("created_at",1)]).limit(3))
result_with_user(early_top)
```

```
Wall time: 107 ms
```

	created_at	text	retweet_count	favorite_count	reply_count	hashtags	name	user_name	verified_status	followers_count	friends_count	statuses
0	2013-06-03 15:32:48	Corona virus....its c...	73776	131225	9968	[]	Marco	Marco_Acortes	False	5652	106	
1	2017-06-08 16:14:48	But to return to a mo...	16	39	0	[]	coldplums	coldplums	False	4341	1435	
2	2020-01-25 18:30:29	I bet the marketing p...	306	797	23	[]	Rick Aaron	RickAaron	False	12391	12854	

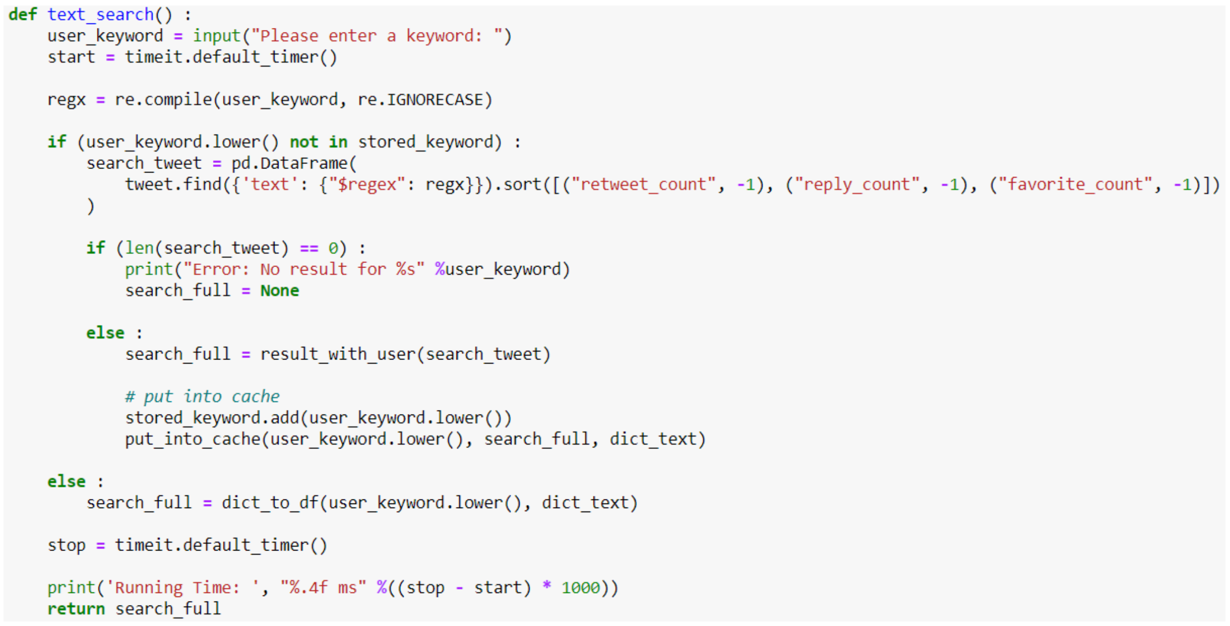
Figure 11. Implementation 5

Of course, in our search application, we offered a user to search a keyword in tweet text, hashtag, or created time. To follow the format of the real Twitter search result, we join the Tweet table and the User table. In the first version, we tried to join the two tables by Pandas and then do searching in the full table which was joining whole tables. However, we discussed it and came to the agreement that it might cost too much running time and space. So, we got the version of the search implementations shown below.

In text search, we used regular expressions to ignore cases, then we found the tweet contains that keyword. To perform relevance sort, we sorted by retweet, reply, and favorite count in descending order. Therefore, it will show the popular tweet at the top. Then, we give this result to the “result_with_user” function to get final merged information to give out to the user.

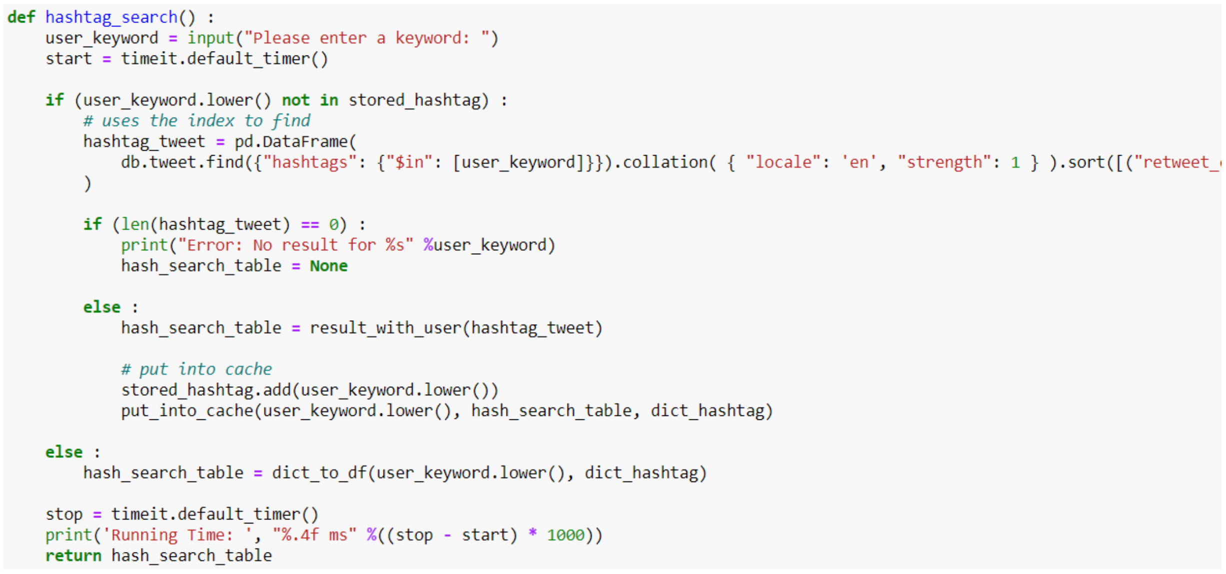
In hashtag search, since *hashtags* is list variable, we used ‘$in’ to determine whether a keyword is in the *hashtags* and used collation to ignore cases. Since we only stored English tweets, the locale is “en” for our design and strength is 1 since we compare the base characters only like ignoring case.

Finally, in date search, we used a while loop to ask for user input again if a user input end date that is earlier than start date. Then, we search the time by $gte and $lte to cover the case when a user searches tweet only for one day.



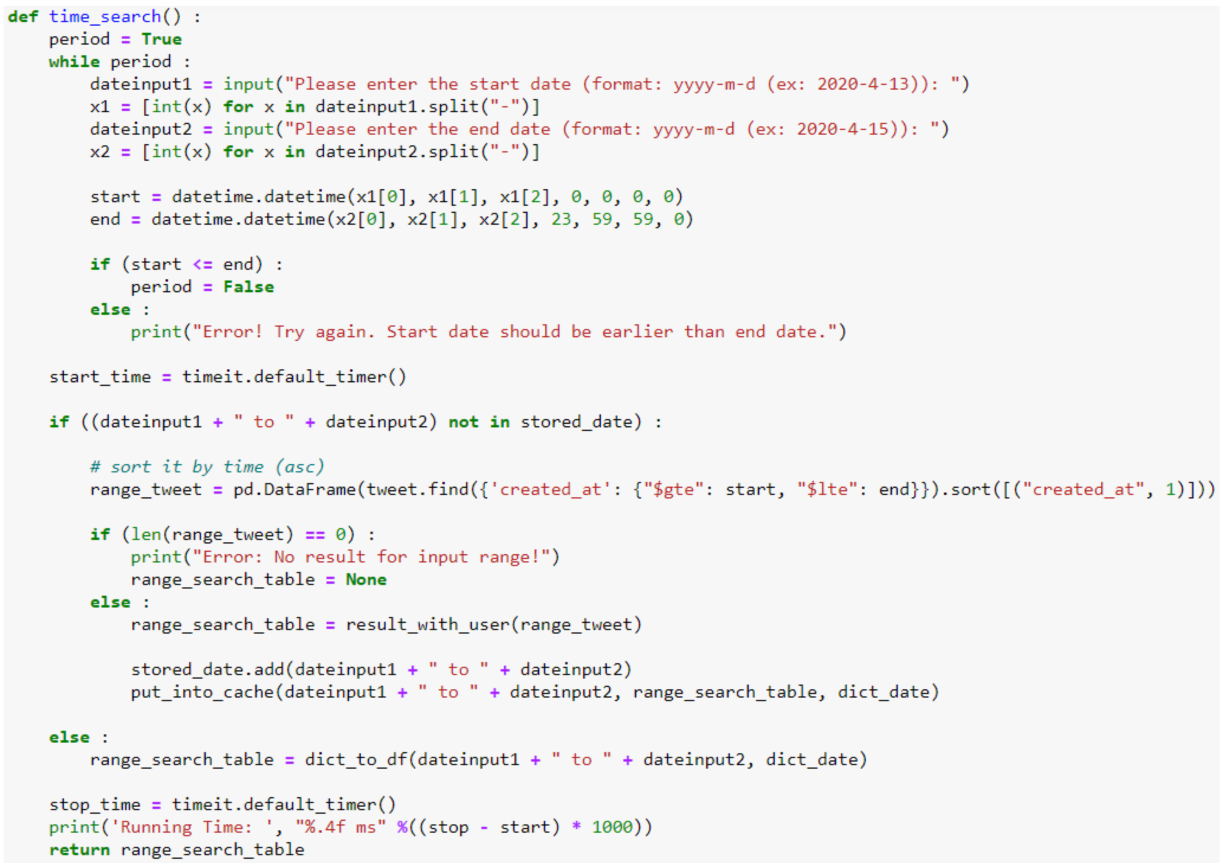
```
def text_search() :
    user_keyword = input("Please enter a keyword: ")
    start = timeit.default_timer()

    regx = re.compile(user_keyword, re.IGNORECASE)

    if (user_keyword.lower() not in stored_keyword) :
        search_tweet = pd.DataFrame(
            tweet.find({'text': {"$regex": regx}}).sort([("retweet_count", -1), ("reply_count", -1), ("favorite_count", -1)])
        )

        if (len(search_tweet) == 0) :
            print("Error: No result for %s" %user_keyword)
            search_full = None

        else :
            search_full = result_with_user(search_tweet)

            # put into cache
            stored_keyword.add(user_keyword.lower())
            put_into_cache(user_keyword.lower(), search_full, dict_text)

    else :
        search_full = dict_to_df(user_keyword.lower(), dict_text)

    stop = timeit.default_timer()

    print('Running Time: ', "%.4f ms" %((stop - start) * 1000))
    return search_full
```

Figure 12. Text search



```
def hashtag_search() :
    user_keyword = input("Please enter a keyword: ")
    start = timeit.default_timer()

    if (user_keyword.lower() not in stored_hashtag) :
        # uses the index to find
        hashtag_tweet = pd.DataFrame(
            db.tweet.find({"hashtags": {"$in": [user_keyword]}}).collation( { "locale": 'en', "strength": 1 } ).sort([("retweet_
        )

        if (len(hashtag_tweet) == 0) :
            print("Error: No result for %s" %user_keyword)
            hash_search_table = None

        else :
            hash_search_table = result_with_user(hashtag_tweet)

            # put into cache
            stored_hashtag.add(user_keyword.lower())
            put_into_cache(user_keyword.lower(), hash_search_table, dict_hashtag)

    else :
        hash_search_table = dict_to_df(user_keyword.lower(), dict_hashtag)

    stop = timeit.default_timer()
    print('Running Time: ', "%.4f ms" %((stop - start) * 1000))
    return hash_search_table
```

Figure 13. Hashtag search



```python
def time_search() :
    period = True
    while period :
        dateinput1 = input("Please enter the start date (format: yyyy-m-d (ex: 2020-4-13)): ")
        x1 = [int(x) for x in dateinput1.split("-")]
        dateinput2 = input("Please enter the end date (format: yyyy-m-d (ex: 2020-4-15)): ")
        x2 = [int(x) for x in dateinput2.split("-")]

        start = datetime.datetime(x1[0], x1[1], x1[2], 0, 0, 0, 0)
        end = datetime.datetime(x2[0], x2[1], x2[2], 23, 59, 59, 0)

        if (start <= end) :
            period = False
        else :
            print("Error! Try again. Start date should be earlier than end date.")

    start_time = timeit.default_timer()

    if ((dateinput1 + " to " + dateinput2) not in stored_date) :

        # sort it by time (asc)
        range_tweet = pd.DataFrame(tweet.find({'created_at': {"$gte": start, "$lte": end}}).sort([("created_at", 1)]))

        if (len(range_tweet) == 0) :
            print("Error: No result for input range!")
            range_search_table = None
        else :
            range_search_table = result_with_user(range_tweet)

            stored_date.add(dateinput1 + " to " + dateinput2)
            put_into_cache(dateinput1 + " to " + dateinput2, range_search_table, dict_date)

    else :
        range_search_table = dict_to_df(dateinput1 + " to " + dateinput2, dict_date)

    stop_time = timeit.default_timer()
    print('Running Time: ', "%.4f ms" %((stop - start) * 1000))
    return range_search_table
```

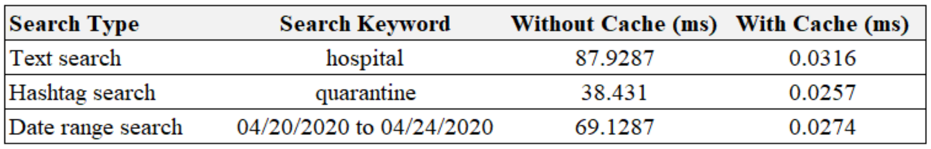
Figure 14. Date range search

Cache Design and Implementation

We used an ordered dictionary of dictionaries to store the keyword and its corresponding dataframe. A Python dictionary is a collection of data values. It is a container that keeps associations between keys and values. The advantage of a Python dictionary is its speed. We can look up a key in a dictionary really fast. We implemented cache for text search, hashtag search and date range search because these are a popular search scenario.

In addition, we stored a keyword that has already been searched in a set. We let a user input a keyword. If a keyword is already in a set, we will return dataframe from a dictionary, otherwise we will perform a search on MongoDB and use result_with_user function to complete the full table result and then we add a keyword into our set and its corresponding data frame into a dictionary. Every time that a keyword that is already in a set is searched, we move this keyword to be the latest one. By using a dictionary, we can implement a cache for many keywords, but we decided to keep only 10 keywords. So, when the 11th keyword is assigned, we will delete the oldest one.

When we tested search queries, it showed that our cache can help optimize running time. However, it uses up a lot of memory space because the Python dictionary takes up more space than other data structures. Also, if we perform dynamic data importing, our cache will be stale. But, in this project, we did not perform dynamic data importing so our cache is not stale.



Search Type	Search Keyword	Without Cache (ms)	With Cache (ms)
Text search	hospital	87.9287	0.0316
Hashtag search	quarantine	38.431	0.0257
Date range search	04/20/2020 to 04/24/2020	69.1287	0.0274

Table 2. Running time of our test search queries with cache and without cache implementation

Conclusion

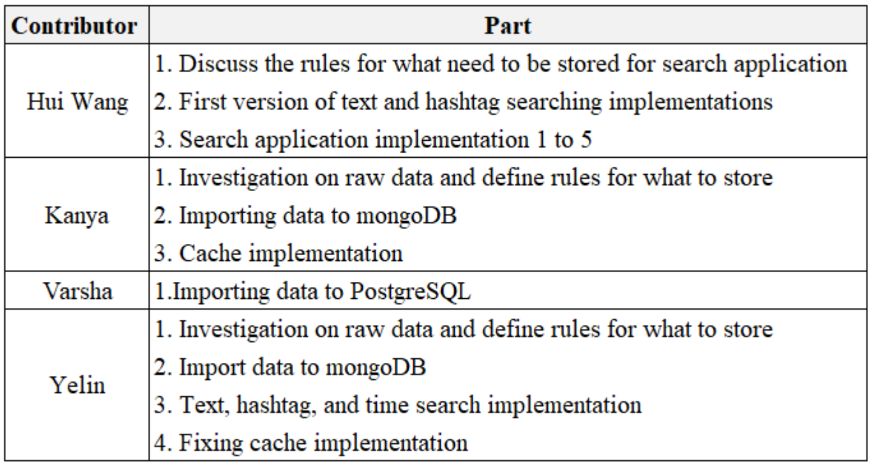
Twitter data is immensely useful in database management applications and can provide vast insights into important campaigns, trending users and topics, public opinion, etc. The complexity of twitter data makes processing, extracting and storing of data an important task in order to ease the building of search applications. We also realized that while setting up a database, it is crucial to make sure that all of the relationships are properly established. For example, having user id as our common parameter, using which we connect the user and tweet table. Finally, we have successfully implemented accessing, pulling and caching of tweet and user information for a few cases in our project.

However, the scope of our project was limited due to the unavailability of Twitter API. Twitter's API can be leveraged in very complex big data problems, involving people or trends. Streaming tweets in real-time would have allowed us to design and test the complex structure of our MongoDB and PostgreSQL databases as well as our search. However, our cache implementation would not be beneficial for this situation because if the new tweet is imported dynamically, then cache will be easily stale. Also, since our topic is a very popular topic in the current situation, it will become stale quickly than other topics. The way we can improve this cache stale issue in our implementation is checking whether the new data contains the keyword in text/hashtag/created_at caches while we import dynamically. If it does, we have to delete the keyword. Then, when the user searches for that keyword, he will get the most updated result. The drawback of this idea is it takes longer time for importing. For popular topics, it is better to not have the cache implementation so it will get the most updated result from the database. On the other hand, if a topic is not popular, it is better to have cache implementation with checking the new tweet whether it contains the cache keyword.

Experience with github

Being inspired by the professor, we decided to use Github for this project. Two of our teammates are the very first time to try to use Github. We communicated and discussed how to use it and how to get used to it. It would always be good to learn from each other. Github is more convenient than we thought, and it saves all the versions of modification. We collaborated 54 commits on Github. Especially during this difficult time given by the long-lasting pandemic, we are glad that there is such an appropriate tool for us to work together, discuss together and code together. Besides, we found that Github Desktop is very helpful and highly recommended to other students in class.

Contribution



Contributor	Part
Hui Wang	1. Discuss the rules for what need to be stored for search application 2. First version of text and hashtag searching implementations 3. Search application implementation 1 to 5
Kanya	1. Investigation on raw data and define rules for what to store 2. Importing data to mongoDB 3. Cache implementation
Varsha	1.Importing data to PostgreSQL
Yelin	1. Investigation on raw data and define rules for what to store 2. Import data to mongoDB 3. Text, hashtag, and time search implementation 4. Fixing cache implementation