Twitter Search Application

Our project goal is to store the twitter data into the databases and build a search application based on the data. We have collected the twitter data using the keyword “U.S. Economy”. We have collected about 16,500 tweets. For collecting tweets we used the api.search twitter API wrapper from the tweepy library. We used this particular wrapper because of its versatility. We had interruption during data downloading, we used max_id to return only statuses with TweetIDs older than previously collected tweets. Our latest data collection was on April 28, 2020. While collecting the data we created a CSV file for PostgreSQL and directly dumped the JSON data into MongoDB.

Data Processing:

Complexity of the twitter data exists in the double nested objects. There are multiple fields of double nested objects such as retweeted status, entities, etc. Hence, we first decided, which information from double nested objects is necessary for our search application and done preprocessing on that. If a tweet is retweeted then we keep the tweet text corresponding to Original tweet only. Apart from that If a tweet is retweeted, We also create keys for ID of original user, Original username, favorite_count of original tweet. For entities nested objects, we created keys corresponding to list of Hashtags, list of mentions, list of media links. Below is the partial screenshots of Preprocessing and collection in Figure 1&2.

```
for tweet in tweet_batch:
        id_ = tweet.id
        id_str = tweet.id_str
        user_id = tweet.user.id_str
        user_name = tweet.user.name
        user = tweet.user._json
        in_reply_to_user_id = tweet.in_reply_to_user_id
        lang = tweet.lang
        source = tweet.source
        retweet_count = tweet.retweet_count
        in_reply_to_status_id = tweet.in_reply_to_status_id
        lang = tweet.lang
        fav = tweet.favorite_count
        try:
            x = tweet.retweeted_status
            retweetedFrom_id = x.user.id_str
            retweetedFrom_OrigTweetid = x.id
            retweetedFrom_name = x.user.screen_name
            RT = True
        except AttributeError:
            retweetedFrom_id = -1
            retweetedFrom_name = 'NA'
            retweetedFrom_OrigTweetid = -1
            RT = False
```

```
if RT:
    content = tweet.retweeted_status.full_text
    retweet_fav = tweet.retweeted_status.favorite_count
else:
    content = tweet.full_text
    retweet_fav = -1
created_at = str(tweet.created_at)
location = tweet.user.location
ht = tweet.entities["hashtags"]
hashtags = []
for i in ht:
    hashtags.append(i["text"]) # List of hashtags.
mntn = tweet.entities["user_mentions"]
mentions = []
for i in mntn:
    mentions.append(i['id_str']) # List of mentions
try:
    md = tweet.entities["media"]
    media = []
    for i in md:
        media.append(i["url"]) # And list of media (pho
except KeyError:  # If no media found
    media = []
```

Figure 1&2: Code used for collection and preprocessing

Data Storage:

Storing 'Tweet' data into MongoDB:

Since not every tweet contains the same structure, tweets data doesn't always make a good .csv file. Also, twitter APIs return tweets encoded as JavaScript Object Notation (JSON) objects and some of its attributes have nested structure. We used MongoDB, an open-source NoSQL database, which simplifies tweet storage, search and recall without the need of writing a tweet parser since it effectively parses the data when we retrieve it. Also, unlike other NoSQL databases, MongoDB offers strong consistency, an expressive query language, and secondary indexes.

MongoDB is a document-based database that uses documents rather than tuples in tables to store data. The documents are BSON objects which are similar to JSON objects using key-value pairs. Due to this similarity, storing, recalling, or searching a tweet in MongoDB is easy but the latter two require an OOP mindset, unlike the traditional SQL command structure.

MongoDB supports query operations that perform a text search of string content. To perform text search, MongoDB uses a text index and the $text operator. Text indexes can include any field whose value is a string or an array of string elements. Use the $text query operator to perform text searches on a collection with a text index. $text will tokenize the search string using whitespace and most punctuation as delimiters, and perform a logical OR of all such tokens in the search string.

We created a test database and created a text index on the content(text of tweet) field to do an experiment to compare the querying time in both scenarios(one with a normal field and search using regex and another a search on text index).

We have done a basic search by word on both the databases and used .explain() function to check the query status. Querying using regex took 16ms whereas the test index took just 6ms. Based on the experiment we have decided to create a text index on the content field. Below you could the snippet of the code and results from Jupyter notebook. We could notice a difference in the resulting number of documents returned which is a result of two different processes used by regex and text index(Which searches on stemmed words).

```
In [49]: tweets.find({"content":{"$regex":"trump","$options" :'i'}}).explain()

Out[49]: {'queryPlanner': {'plannerVersion': 1,
           'namespace': 'TWT_DB.tweets',
           'indexFilterSet': False,
           'parsedQuery': {'content': {'$regex': 'trump', '$options': 'i'}},
           'winningPlan': {'stage': 'COLLSCAN',
            'filter': {'content': {'$regex': 'trump', '$options': 'i'}},
            'direction': 'forward'},
           'rejectedPlans': []},
          'executionStats': {'executionSuccess': True,
           'nReturned': 2240,
           'executionTimeMillis': 16,
           'totalKeysExamined': 0,
           'totalDocsExamined': 15429,
           'executionStages': {'stage': 'COLLSCAN',
            'filter': {'content': {'$regex': 'trump', '$options': 'i'}},
            'nReturned': 2240,
            'executionTimeMillisEstimate': 0,
            'works': 15431,
            'advanced': 2240,
            'needTime': 13190
```

Figure 3: Querying using regex

```
In [52]: test.find({"$text":{"$search":"trump"}}).explain()

                'isSparse': False,
                'isPartial': False,
                'indexVersion': 2,
                'direction': 'backward',
                'indexBounds': {}}}}}},
           'rejectedPlans': []},
          'executionStats': {'executionSuccess': True,
           'nReturned': 1702,
           'executionTimeMillis': 6,
           'totalKeysExamined': 1702,
           'totalDocsExamined': 1702,
           'executionStages': {'stage': 'TEXT',
            'nReturned': 1702,
            'executionTimeMillisEstimate': 0,
            'works': 1703,
            'advanced': 1702,
            'needTime': 0,
            'needYield': 0,
            'saveState': 13,
            'restoreState': 13,
            'isEOF': 1
```

Figure 4: Querying employing text index

Storing 'user' data into PostgreSQL:

We converted the 'CSV' twitter data file into Pandas DataFrame. From this, we extracted the 'user' column which contained strings of dictionaries and created a new dataframe with its entities as 'columns'. Twitter no longer supports many of the 'user' object attributes. We dropped them from our dataframe. We renamed 'id' as 'user_id' to distinguish it from the 'id' in the 'tweet' object.

We chose Postgres as our Relational Database and used pgAdmin as a user interface. We created a database in Postgres to store our 'user' data. For interacting with our database directly from Python, we used SQLAlchemy which provides us the benefit of working with tables and queries as objects in Python. We installed a driver named psycopg2 (pip install psycopg2) for facilitating SQLAlchemy to obtain a connection to Postgres. Then, we created an engine, using the connection string of our database, with which SQLAlchemy can create tables in our database.

Our next step is to remove the duplicate rows in our 'user' data. We obtained the tweet's created_at column and inserted it into our 'user' data. Using SQL's window and aggregate functions, we retained only the most recent data of all users in our data. We then extracted it into a dataframe and replaced it with our old 'user' data. Index for our 'user' data is 'user_id'.

Removing duplicate rows in user_df:

```
new_user_df = pd.read_sql_query('''WITH t1 AS (SELECT *,
                            RANK() OVER(
                            PARTITION BY user_id
                            ORDER BY c_at DESC
                            ) rank_created
                            FROM user_df)
                SELECT *
                FROM t1
                WHERE rank_created = 1''', con = engine)
```

Figure 5: Removing duplicate rows from 'user' data

Search Application

With the Tweet table and the User table stored, we designed a search application on the information we have. We used pymongo for the searching in the Tweet related data from MongoDB, psycopg2 for the searching in the User related data from postgreSQL, and Pandas to link the two tables and show the result.

In our search application, we offered the user to search by:

- Keyword
- Hashtag
- Created time
- User_id
- User_name

- Keyword, hashtag and threshold on number of followers
- Keyword, hashtag and a location.

We have also given the user a few other options like “Tweet with maximum retweet_count”, “Top 10 hastags” etc.

In search by keyword, we used a text index. In hashtag search, since hashtags is list variable, we used ‘$in’ to determine whether a keyword is in the hashtags. In date search, we used $gte and $lte to cover the time range given by the user.

We have designed a basic web interface for the above mentioned queries and below please find the snippets of web interface showing the results of few search queries.

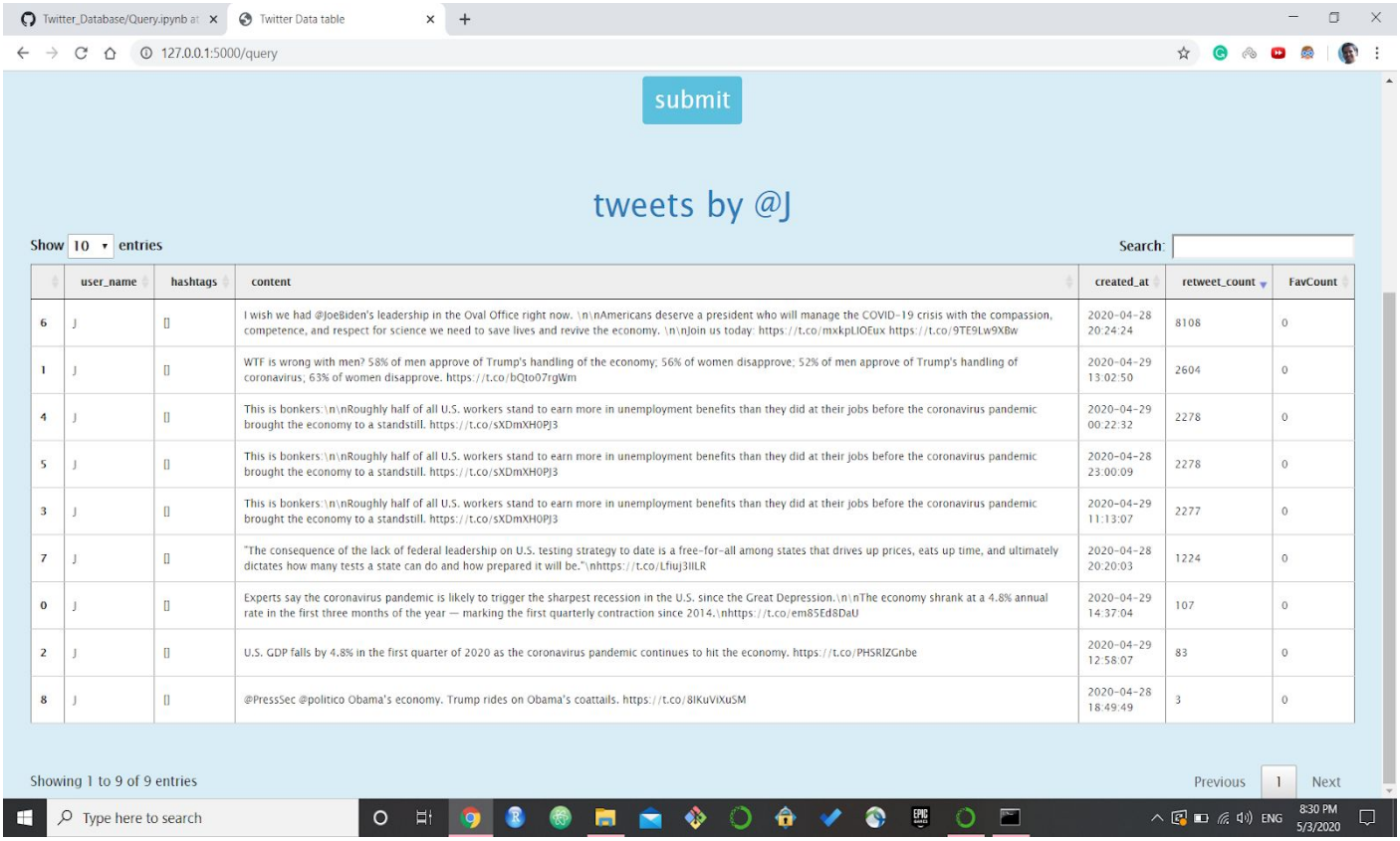

Figure 6: Search application queries

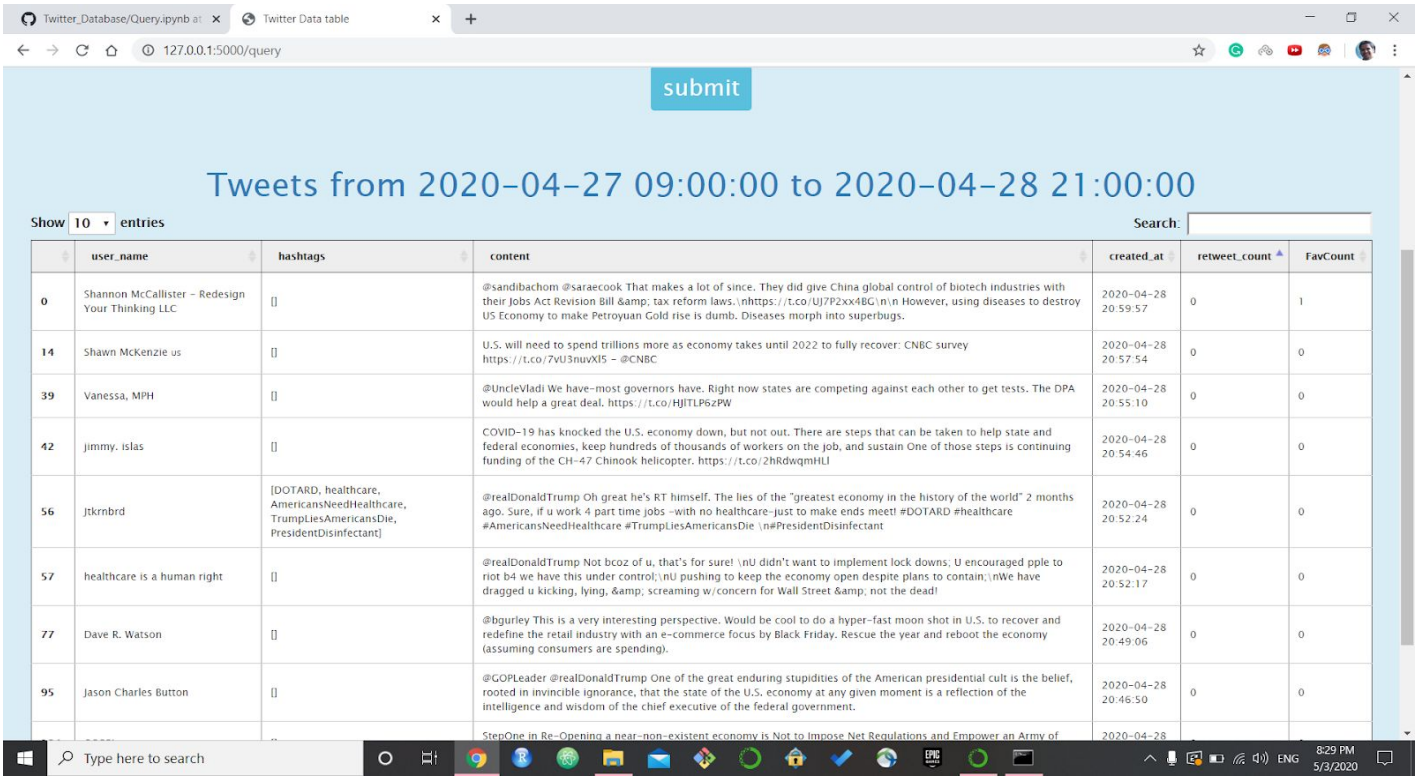

Figure 7: Searching the tweets in a time range

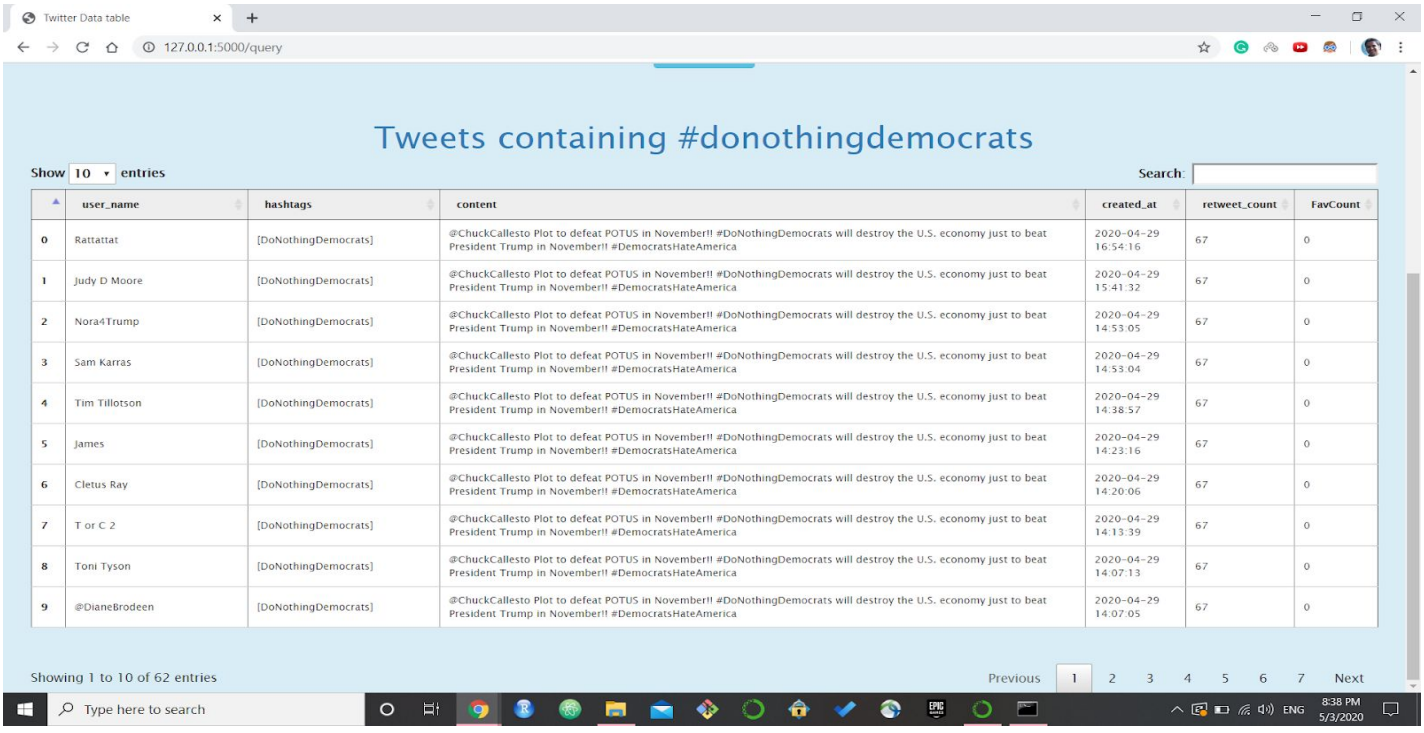

Figure 8: Searching by hashtags

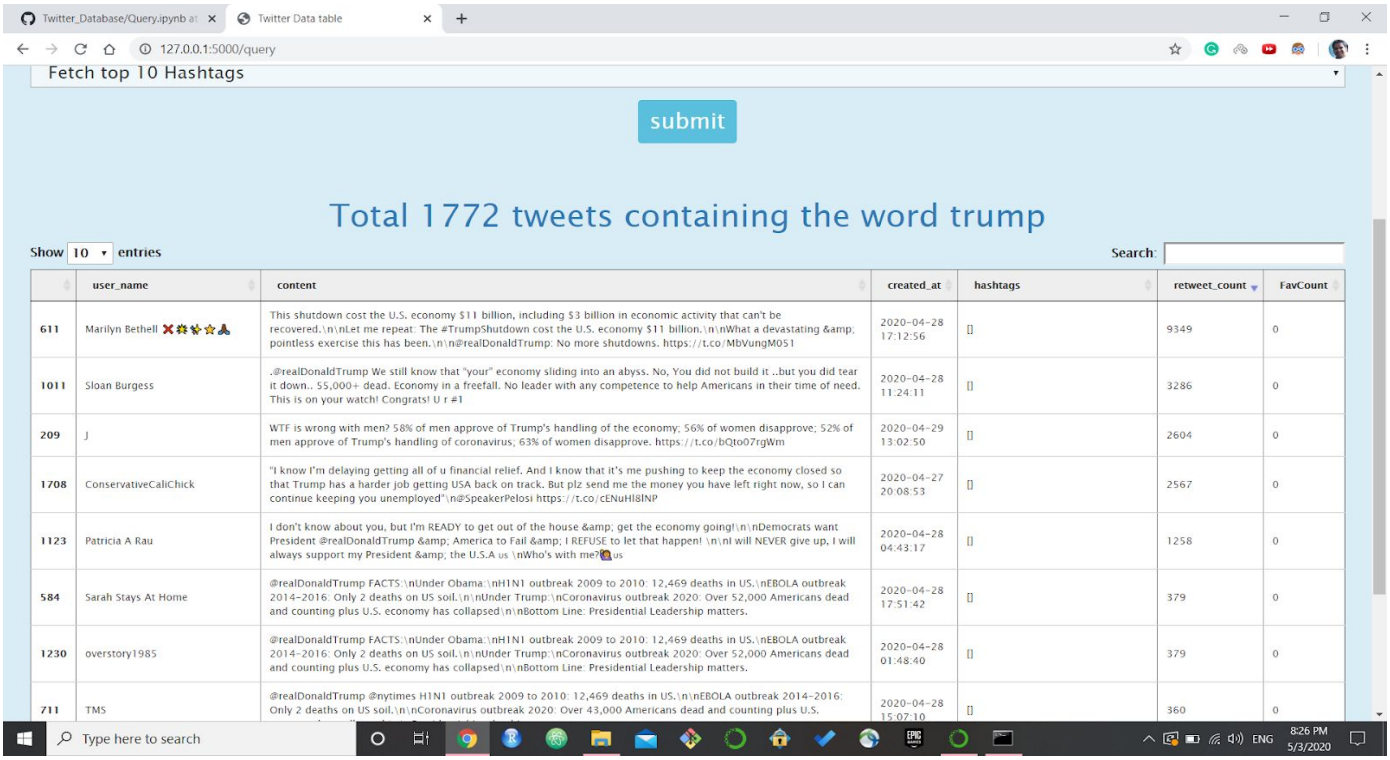

Figure 9: Searching the tweets by word

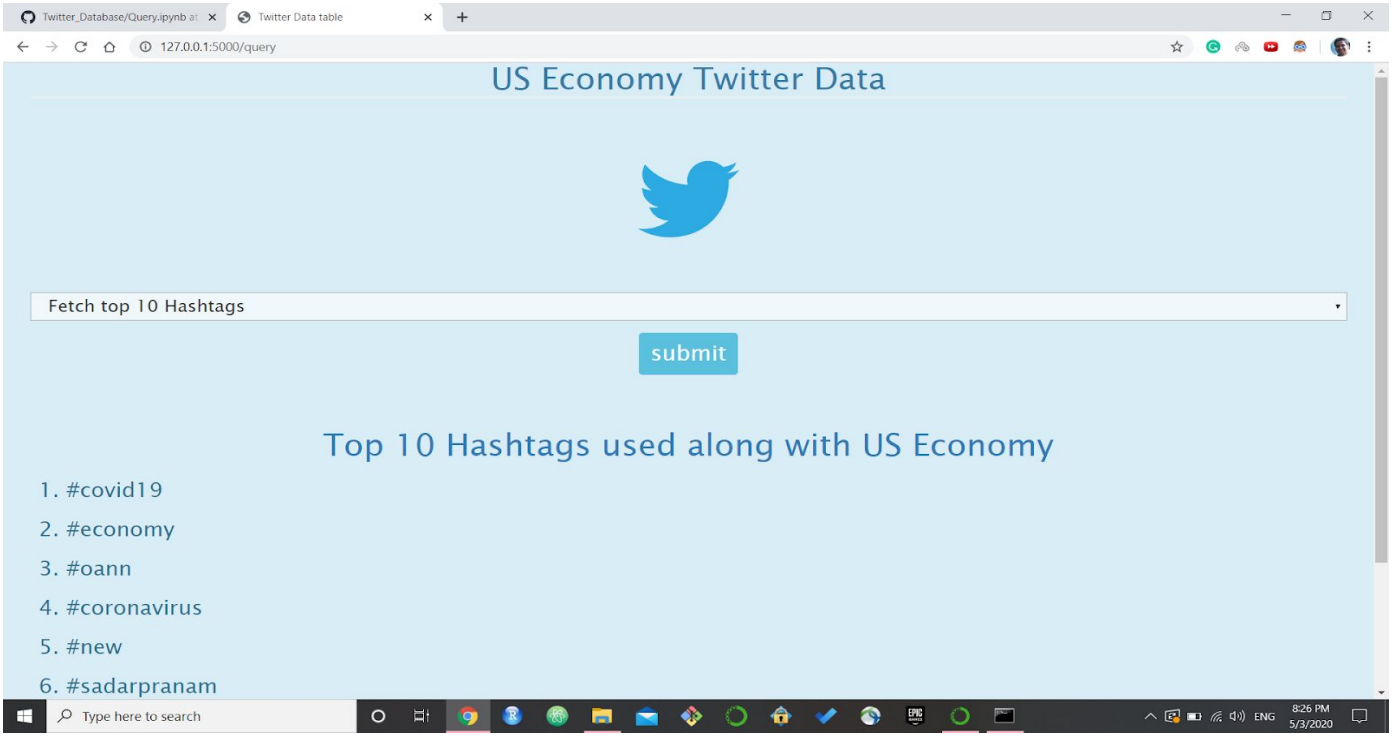

Figure10: Fetching 10 most used hashtags

Caching With Redis:

Performance is the most important evaluation metrics for our twitter search application. When performing operations on our search application which interact with PostgreSQL and MongoDB database, some queries are the bottleneck. As our MongoDB database scales up some of the queries can be really slow down. This issue became more serious when there is a connection issue between the search application server and database server. Hence, caching is really important to overcome these issues.

Caching is used to overcome the database storage problem. As our storage in a database increase, query running time increases. Databases are larger in size but are relatively slow in querying compared to the Random Access Memory(RAM). Hence, cache is used to store recently accessed data in the Faster storage system. We used Redis in-memory data structure which is really fast as compared to MongoDB. Each time when the query is run, we first check if query results are available in the cache. If the results are not stored in the cache we will get the result from the database and saved the results to the cache(Figure11).

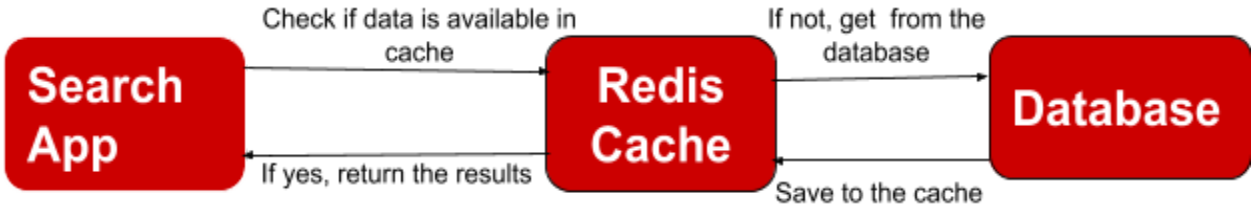

Figure 11: Caching with Redis

When implementing cache it is often handy to to automatically evict the old data as new data is added. Redis uses LFU(Least Frequently Used) eviction policy for this purpose. LFU tracks the frequency of access of queries, so that the ones used rarely are evicted while the one used often have a higher chance of remaining in memory.

Our implementation of cache: First we configure our Redis and database connection. It is necessary to connect with Redis, MongoDB, and PostgreSQL connection while searching for a query. Hence, we created a function for the same and you can see that in Figure12.

It is necessary that Redis Cache should be updated on a configurable schedule. We cannot keep our Cache the same and independent from our database. Hence, we keep configurable parameters for cache update and the default value is one month. In the below Figure11 you can see the database connection and Redis connection. You can also see the default value for cache expiration.

```
In [2]: MAX_EXPIRE_DURATION = 24 * 3600

In [3]: def configure(max_redis_connections = 2):
            client = pymongo.MongoClient()
            globals()['client'] = client
            cache_pool = redis.ConnectionPool(host = 'localhost', port = 6379, db = 0, max_connections = max_redis_connection
            globals()['cache_pool'] = cache_pool

In [4]: configure()

In [5]: def get_database_connection():
            '''
            This method should be called to get a connection to the MongoDB server
            '''
            return globals()['client']

In [6]: def get_cache_connection():
            '''
            This method should be called to get a connection to the Redis server
            '''
            return redis.StrictRedis(connection_pool = globals()['cache_pool'])
```

Figure12: setting up a connection with Redis and Database

Next, We created a function to search for a query. Its first check if results are available in Cache using a query as a key and if not it searches in the database otherwise retrieves from the database. You can see that function below in Figure13.

```
def search(query):
    '''
    The function perform the query search.
    It first checks the cache to see whether the key exists.
    If yes, then the query result corresponding to cache are retrieved from Redis,
    else an query is fired to database.
    '''
    cache = get_cache_connection()
    if cache.exists(query):
        print('Exist in Cache')
        results = lookup_cache_key(query)
        return(results)
    else:
        print('Do not exist in Cache')
        results = database_lookup(query)
        return(results)
```

Figure 13: The main function for search and checking if results available in Cache

We retrieve results from the cache with the corresponding key, as shown in below Figure 14.

```
def lookup_cache_key(cache_key):
    '''
    Lookup a key in Redis cache
    '''
    cache = get_cache_connection()
    ids = cache.get(cache_key)
    return ids
```

Figure 14: retrieving results from the cache using the corresponding key

Otherwise, we retrieve results from the database and save the results in the cache. We also save the corresponding expiration time for the cache. By default expiry time of the cache is 1 month(Figure15).

```
def database_lookup(query):
    '''
    This will run when cache is not available
    This function will search the results in MongoDB
    and saved the cache to the Redis
    '''
    myquery = {"content":{"$regex":query,"$options" :'i'}}
    client = get_database_connection()
    db = client["tweet_database"]
    tweets = db.tweets_collection
    twts = list(tweets.find(myquery))
    save_query_to_cache(query, twts)
    return twts
```

```
def save_query_to_cache(cache_key, twts):
    '''
    Save the query result of into Redis cache
    Also, given the expiry date for the duration
    '''
    cache = get_cache_connection()
    serializedObj = dumps(twts)
    cache.set(cache_key, serializedObj)
    cache.expire(cache_key, MAX_EXPIRE_DURATION)
```

Figure 15: retrieving results from the database and saving in the cache

If we run the same query again we get different timing. First, it retrieves results from the database while the second time it retrieves from Redis.

```
## Run the query first time
import time
start_time = time.time()
search("covid19")
print("--- %s seconds ---" % (time.time() - start_time))
```

```
Do not exist in Cache
--- 0.22595620155334473 seconds ---
```

```
## Run the same query again, change the time difference
import time
start_time = time.time()
search("covid19")
print("--- %s seconds ---" % (time.time() - start_time))
```

```
Exist in Cache
--- 0.003137350082397461 seconds ---
```

Figure 16: Running the same query simultaneously to check cache functionality.

Lastly, we compare the performance of our application with Cache hit and without cache hit(Table1) for different queries.

Query	Timing without Cache	Timing with Cache
Tweets with the word "Covid19"	113 Millisecond	3.3 Millisecond
Tweets with the hashtag "#Trump"	157 Millisecond	3.5 Millisecond
Search tweets for particular User	21 Millisecond	2.7 Millisecond

Table1: Performance comparison with Cache

Web User Interface:

We used Python flask the micro-web framework for connecting python with the user interface. For the UI we used technologies like Bootstrap, jQuery, and CSS, along with HTML. For passing the dataframe and the variables from the back-end, we used Jinja2. To divide the lengthy table into small chunks of rows per page, we used Data tables plugin.

Github for collaboration:

We managed our project using Github. Though previously some of us frequently used GitHub to just track personal projects and showcase our work. This is the first time we used GitHub for a team project. Many of us were working on the same functionality and hence the same file. Hence, the merging facility of GitHub was really great to merge our work. Also because of the team project, it was hard and confusing to keep the track revisions i.e. who changed what and when. GitHub took care of this problem by keeping track of all the changes that have been pushed to the repository. We collaborated on a total 59 commits on Github. Github was really helpful especially in this pandemic time for collaborating and managing our project.

Scalability for real-time streaming with BigData:

Although MongoDB was designed to be fast and to have a great performance with unstructured data, relational databases like PostgreSQL can have better performance as the data size increases. Also, they use dynamic and static schemas that help us to link data when they are relational. Furthermore, the advantages of using a relational database are data integrity and avoiding data redundancy.

In our dataset, we had around 14,400 retweeted tweets and 2000 original tweets. Most of the retweeted tweets were from a few famous twitter accounts. Hence, most of our dataset was redundant. If we keep all the unique retweets into the PostgreSQL database and join it with the user, we can save a lot of memory. This is really important as our data size increases. If needed we can send the result of the joined table to MongoDB for text processing.

For our short application we have used MongoDB, PostgreSQL, Redis and Flask webapp. We have to manage all this platform manually and maintain the connection between each of these applications. It gets more complicated as our data size increases and as we start working with live streaming. Also, we might integrate more tools as our users as well as data increases. That's when it became necessary for us to use the Apache kafka for data movement from one application to another. We will use Twitter API as the producer, MongoDB and PostgreSQL as consumer as well as producer. Lastly, our Flask webapp as a Consumer.

In conclusion, we will primarily use PostgreSQL for data storage and MongoDB for data operations. Finally, Apache Kafka for data movement.

Conclusion:

Twitter data is complicated when compared to facebook or instagram or other social media. On twitter we have retweets, replies(Which are comments but its a tweet on its own) and quoted tweets. This complexity of twitter data makes the design of storage databases an important task in order to ease the processing, querying of the stored data. We realized the importance of structural elements of databases like indexes(querying time depends on it), the need for splitting data into different tables(This is done for easy querying, saving the disk space in the database), and also we noticed it is crucial to make sure that all the relationships between the databases/tables are properly established. For example, we have user id as our common parameter, using which we connect the data from postgreSQL and MongoDB and it was crucial to have both the columns with the same data types.

LFU(Least Frequently Used) eviction policy for caching is really helpful as the users of applications increases and if the application deployed on real time streaming. Because LFU caching mechanism will track the frequency of requested queries. LRU cache implementation has certain

drawbacks, Suppose if an item is recently accessed by a certain user but if it's never requested the same query again, so that query will not get expired. There might be the case that certain queries which are frequently accessed might get evicted due to infrequent queries(as we get out of RAM). Hence, it is important to store only frequent queries and not all. LFU(does the exact same), as the number of users increase we might have in-memory limitations hence its important to store cache based on Frequency. Hence, LFU is better than LRU for scalability.

Overall, working on this project helped us learn the basics of database management systems and in particularly the structure and usage of MongoDB, postgreSQL databases, Redis caching, and Flask WebApp. We have also learnt the importance of teamwork and coordination to build a successful DBMS and also the need for application/tools like github.

Contributions:

Name	**Work**
Sanket	Data Downloading and Processing, Redis Caching, Pymongo Queries
Naren	WebApp using Flask, Redis Caching
Tejeshwar	Pymongo Queries, PostgreSQL Queries, WebApp using Flask
Hemachandra	PostgreSQL Queries, Data Processing, Integration of PostgreSQL and MongoDB