CS 242 Project Report phase 1

Members:

Madhusudhan Tungamitta(862325125)
Balaji Arunachalam(862321425)
Shreya Godishala(862313765)
Imran Shahid (862319859)
Thirumalai Vinjamoor Akhil Srinivas(862320790)

Language Used: Python, Java (Maven)
Api Used: Tweepy and Lucene

Requirements to Run Twitter crawler :
Python 3.
Install Tweepy
Java installed

Instruction For Running Twitter Crawler and Lucene:

1. Edit the crawler.bat file with the location of your local system and run the crawler.bat file, it will automatically get executed if python has been installed.
2. Extra Credit thing: we have added another folder with files that handle Location and Emoji. This python file handles the emojis and pulls tweets with geolocation in one file, and pulls non-geolocation tagged tweets in another file.
 2.1. Edit the crawler.bat file with the location of your local system and run the crawler1.bat then it will generate the two CSV one with location and another without location to store the tweets and images will be downloaded under the images folder.
3. Lucene:
   a. We first install all the files by importing the project in Eclipse and then running it as a maven clean then a maven install.
   b. Then we navigate to the project properties (right-click) and change the compiler to release version with default compiler settings. Then run as a Java application.
   c. Set the path as needed in the indexer for the data.

Contributions

We are 5 people working on this project. For the first phase of the project, two-part of the project is split into 2 people for Twitter Crawler and 3 people for Lucene.

Twitter Crawler:
Balaji Arunachalam(862321425) and Madhusudhan Tungamitta(862325125) has worked on this part using the Tweepy library provided by Twitter.

Lucene Indexing:
Imran Shahid(862319859), Thirumalai Vinjamoor Akhil Srinivas(862320790), Shreya Godishala(862313765) has worked on the part of the indexing using java

Why twitter:
Twitter is an excellent source of data. It gives us the freedom to crawl the data with the official API they are providing. We are using Tweepy in this project for crawling the data. The API class provides access to the entire Twitter RESTful API method. Each method can accept various parameters and return responses.

We are using the Twitter developers essential with the help of it we are able to retrieve the data up to seven days back.

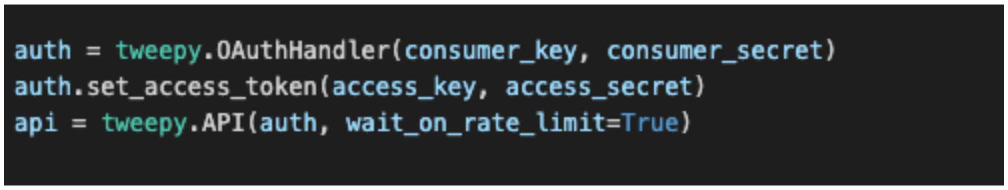
Twitter Crawling Architecture:

Our architecture consists of two files one is config.py and the other is twittercrawler.py

Config.py:
Config files contain the API keys/consumer key, API/Consumer Secret, Access key, Access key secret that is generated from the app we have created in Twitter. we created the app in the Twitter developer account is the first task we did in this project.

Twittercrawler.py:
The maximum number of requests that are allowed is based on the time interval, some specified period, or window of time. The request limit interval is fifteen minutes. If an endpoint has a rate limit of 450 requests per rate limit window (OAuth2) / 900 requests/ 15- minutes, then up to 900 requests over a 15-minute interval is allowed(OAuth1).

As we are crawling over 1 GB of data, we need to request more than 900 within 15 minutes. Hence, we have added a wait-on rate limit. It helps us to retrieve data automatically. The crawler will go to sleep more on the limit of 900 requests per 15 minutes in reach. After 15 minutes our code will again automatically start the crawling process the



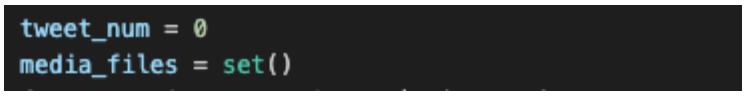
```
auth = tweepy.OAuthHandler(consumer_key, consumer_secret)
auth.set_access_token(access_key, access_secret)
api = tweepy.API(auth, wait_on_rate_limit=True)
```

we have crawled the data based on geolocation and if geolocation is present, we are crawling and storing the data.

Twitter Crawling Strategy

Handing Duplicate Tweets: -

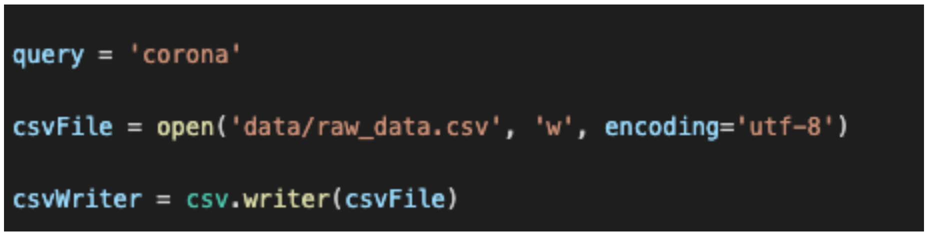
Using the hash set we are retrieving the data Since the set does not allow the duplicates it helps in reducing the extra efforts of removing duplicate data



```
tweet_num = 0
media_files = set()
```

Data Storage:

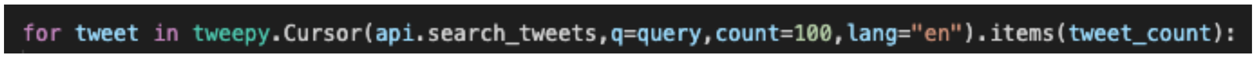
We are writing our tweet data in a CSV writer and writing the text file in the CSV file using UTF-8 encoding as it contains the special characters. and



```
query = 'corona'

csvFile = open('data/raw_data.csv', 'w', encoding='utf-8')

csvWriter = csv.writer(csvFile)
```

Tweepy Cursor:

The cursor method is the latest release of Tweepy. It handles the pagination work for us behind the screen so our code can now just focus on processing the results.

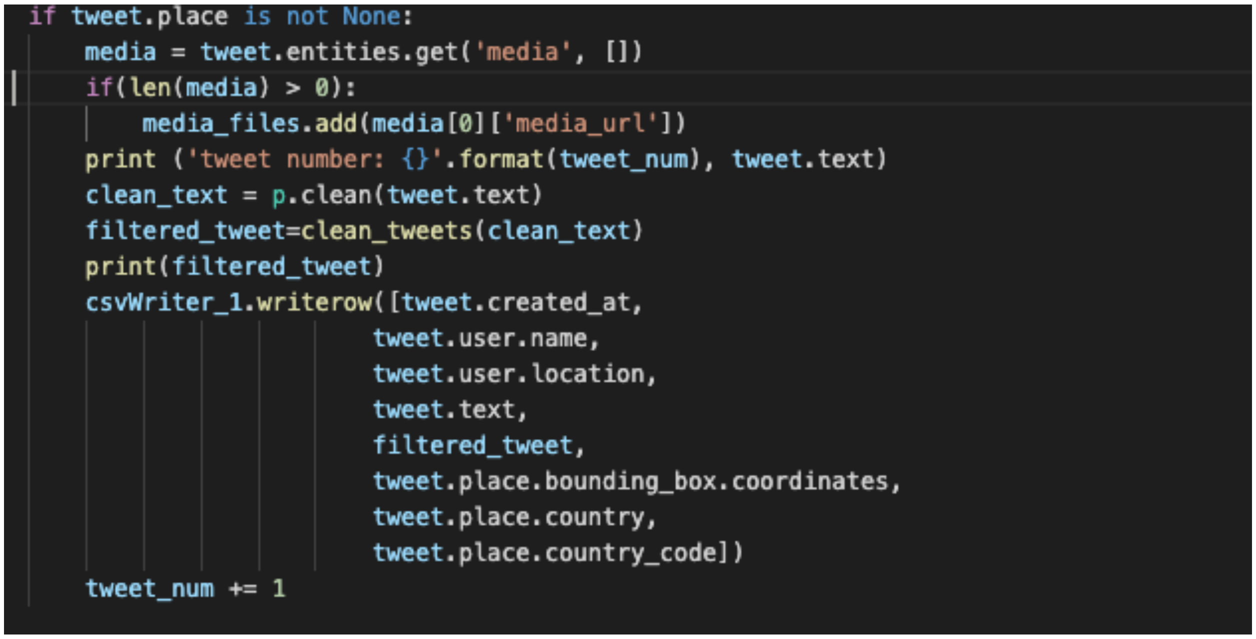


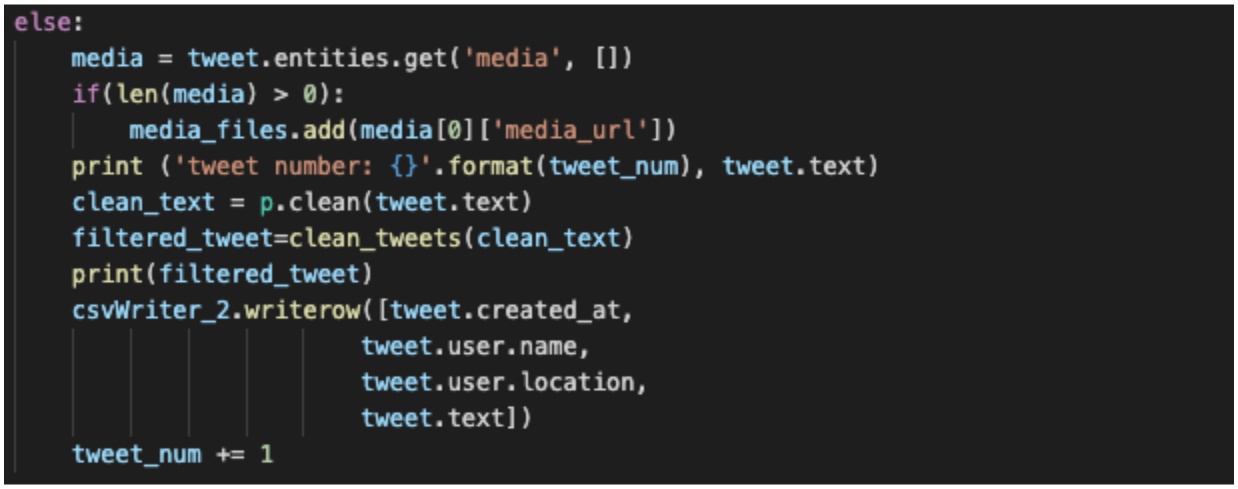
```
for tweet in tweepy.Cursor(api.search_tweets,q=query,count=100,lang="en").items(tweet_count):
```

We have passed our parameters like our query, count, and the language into a cursor, and it will automatically pass the parameters into request whenever we make a request.

Tweet_count in the items () is the total number of tweets users need to pull so it has to be passed as an argument while executing the python file, here we have passed in the argument inside the bat file.

Location and Emoji's Handling Folder:
As we are planning to pinpoint the location for Hadoop and show the pinpoint in the retrieving results, we have created another folder (Location and Emoji Handling) we are filtering the data based on whether the place is found or not in the tweet. If we found the location then write the user location, followers count ……. Data is stored in data tweet_pulled_with_location.csv
If we don't find any location, we write the username, follower count …. Etc in the tweet_pulled_without_location.csv writer.


```python
if tweet.place is not None:
    media = tweet.entities.get('media', [])
    if(len(media) > 0):
        media_files.add(media[0]['media_url'])
    print ('tweet number: {}'.format(tweet_num), tweet.text)
    clean_text = p.clean(tweet.text)
    filtered_tweet=clean_tweets(clean_text)
    print(filtered_tweet)
    csvWriter_1.writerow([tweet.created_at,
                          tweet.user.name,
                          tweet.user.location,
                          tweet.text,
                          filtered_tweet,
                          tweet.place.bounding_box.coordinates,
                          tweet.place.country,
                          tweet.place.country_code])
    tweet_num += 1
```



```python
else:
    media = tweet.entities.get('media', [])
    if(len(media) > 0):
        media_files.add(media[0]['media_url'])
    print ('tweet number: {}'.format(tweet_num), tweet.text)
    clean_text = p.clean(tweet.text)
    filtered_tweet=clean_tweets(clean_text)
    print(filtered_tweet)
    csvWriter_2.writerow([tweet.created_at,
                          tweet.user.name,
                          tweet.user.location,
                          tweet.text])
    tweet_num += 1
```

LUCENE

Fields in the index

We have two files tweetParser.Java, tweetIndexer.Java

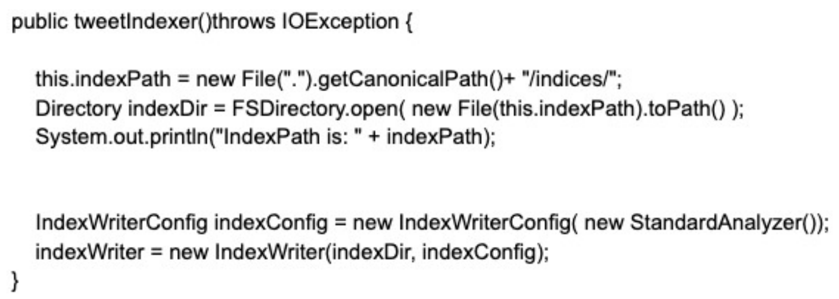
TweetParser.Java

The parser has one purpose and that is to set the structure of the tweet to be extracted and to parse and break the JSONObject into smaller parts and then pass it to the document function to be added to the index.

TweetIndexer.Java

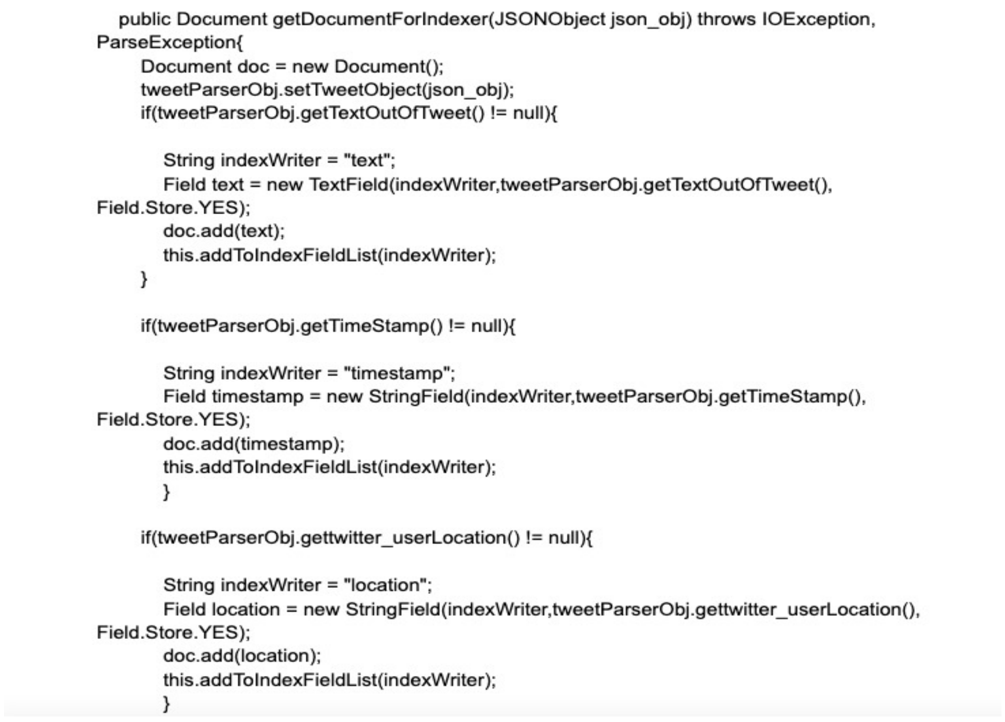
We are creating the indexer using the new indexer() command and then We are parsing the JSON object in order to analyze the string that is passed

The fields used are the coordinates given as coords, The location given as location. Username given as user. The tweet data itself is given as text and the date and time given as such.The indexing strategy is to index by the text and geospatial data that exists within the tweet. To this end, we have used a few different types of text analyzers and indexing techniques.



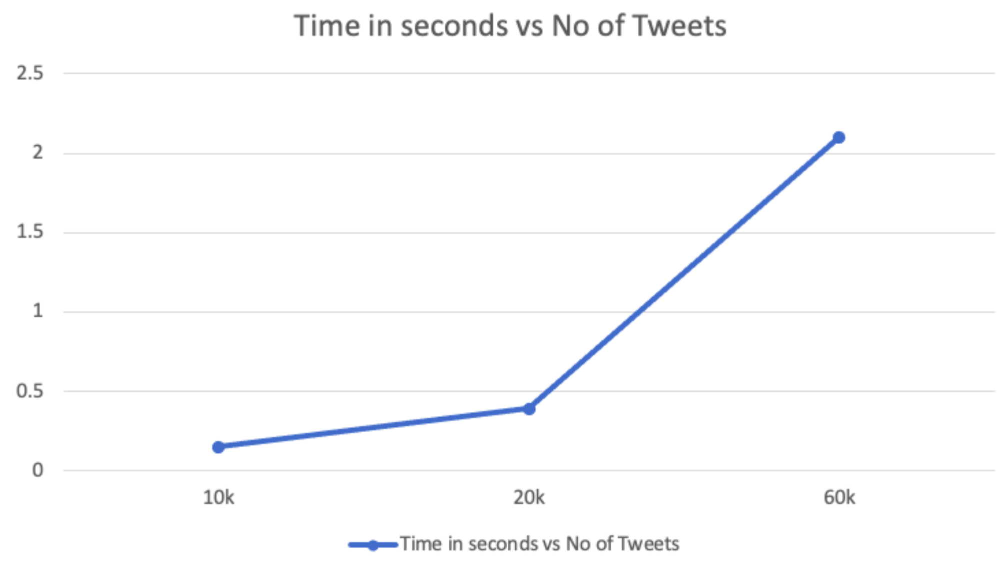
```
public tweetIndexer()throws IOException {

    this.indexPath = new File(".").getCanonicalPath()+ "/indices/";
    Directory indexDir = FSDirectory.open( new File(this.indexPath).toPath() );
    System.out.println("IndexPath is: " + indexPath);


    IndexWriterConfig indexConfig = new IndexWriterConfig( new StandardAnalyzer());
    indexWriter = new IndexWriter(indexDir, indexConfig);
}
```

Above is the index creation function called in the main, it uses the StandardAnanlyser for text and strings. It writes the indexes to the new index writer,



```
    public Document getDocumentForIndexer(JSONObject json_obj) throws IOException,
ParseException{
        Document doc = new Document();
        tweetParserObj.setTweetObject(json_obj);
        if(tweetParserObj.getTextOutOfTweet() != null){

            String indexWriter = "text";
            Field text = new TextField(indexWriter,tweetParserObj.getTextOutOfTweet(),
Field.Store.YES);
            doc.add(text);
            this.addToIndexFieldList(indexWriter);
        }

        if(tweetParserObj.getTimeStamp() != null){

            String indexWriter = "timestamp";
            Field timestamp = new StringField(indexWriter,tweetParserObj.getTimeStamp(),
Field.Store.YES);
            doc.add(timestamp);
            this.addToIndexFieldList(indexWriter);
            }

        if(tweetParserObj.gettwitter_userLocation() != null){

            String indexWriter = "location";
            Field location = new StringField(indexWriter,tweetParserObj.gettwitter_userLocation(),
Field.Store.YES);
            doc.add(location);
            this.addToIndexFieldList(indexWriter);
            }
```

Above is the getDocument function which we use to extract the parsed Tweets from the Json objects passed to TweetParser. Here we extract the username, location, timestamps and text data from the tweet itself.
Given below is a graph of the time taken to index the tweets by the number of tweets mentioned, we stress test the system by checking it against 10, 20, and 60 thousand tweets.



The limitations of the system are such. We do not index hashtags or media data directly but rather index the URLs removing some semantic inference, and the data must be input sequentially and long read times may cause issues with production values of the model.