Big Data Algorithms - CMPE-297

Project Report

Twitter Search Engine

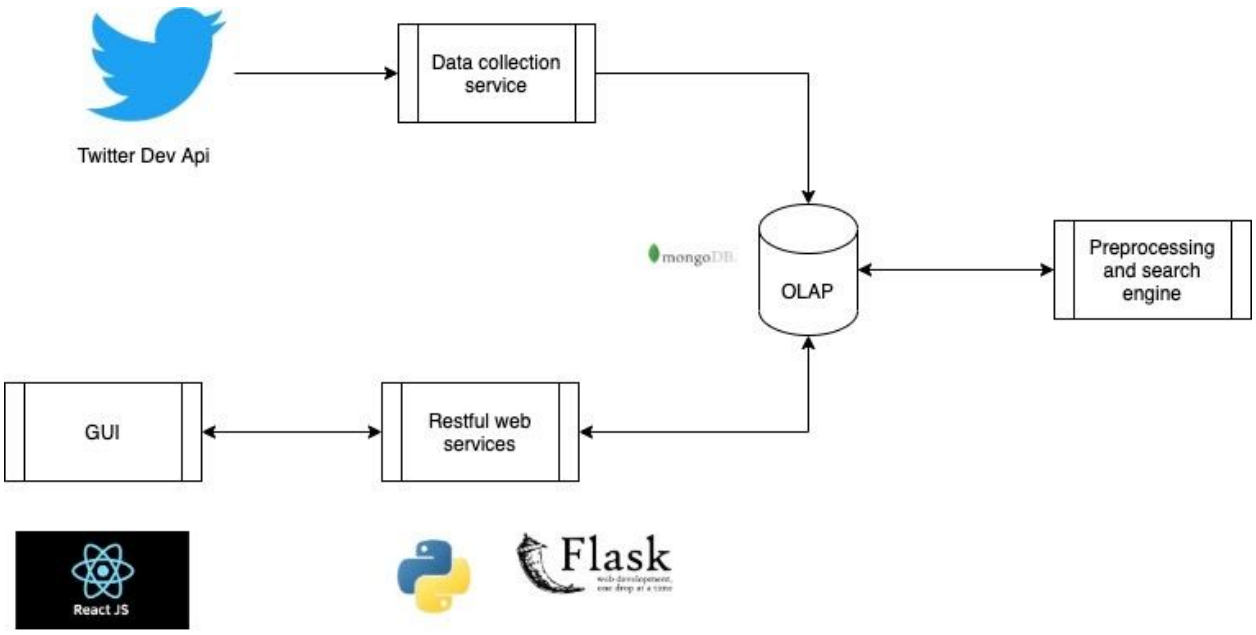
Group Members:

Anurag Patro, Purvi Misal, Yashwant Khade

1 . Project Description

We plan to develop a search engine for tweets for the topic sports. We aim to do so by using the developer portal of Twitter, MongoDB database, and some indexing algorithms. In order to create a database, we develop a pipeline to collect the required tweets from the Twitter developer API and store them into MongoDB. Once we have our database ready we implement a robust engine that could parse the tweets, process them, and index them for future queries. To process text queries, we will be creating tf-idf weights for all words of all the documents/tweets and then using similarity/distance measures to compute the tweets ranked from most appropriate to least. We will also be indexing our MongoDB collection on the hashtags, locations, etc. for different types of searches. We are designing restful web services to code the functionality and logic of our application. Our app will also have an apt UI to enhance the look of query searches and results. The end-user will get insights about different tweets based on location, users, hashtags, and as well as different phrase queries. We also aim to provide analytics for different approaches and how we could optimize our search engine

2. Architectural Overview



Data Pipeline:

This part of the project is responsible for collecting Twitter data in MongoDB database. This consists of two steps:

1. Streaming tweets
   In this step, we used the Tweepy Python library to scrape data from Twitter, which is a wrapper for Twitter's Developer API and has appropriate classes to build a custom stream listener. After building a custom modified Stream Listener which stores tweets into JSON file, we created an object of OAuthHandler and StreamListener to stream tweets of sports-related topics.
   For example,
   track = ['sports', 'cricket', 'ipl', 'India vs Australia', 'BCCI', 'IPL2020', 'ICC', 'SportsCenter"]
2. Storing tweets
   In this step, the intermediate JSON file is read and only the relevant information from the tweet JSON object is relayed into the MongoDB collection. We kept only the information that was relevant to our problem statement of Search Engine like user name, tweet text, hashtags, location, etc. which are fields that are relevant for searching and dismissed information like user details, user-created at, followers, following, etc. The final tweet document that gets stored in DB collection is described in the next section.

Search Engine:

As the data is received in MongoDB it is used by the search engine service for preprocessing and developing TF-IDF matrix.

Restful Web services:

We have two services that handle Search Text and Search On Fields.Search Text method handles searching on text using the TF-IDF vector and Search on Fields method handles point queries on 4 different fields of a tweet document.

3. Database Details

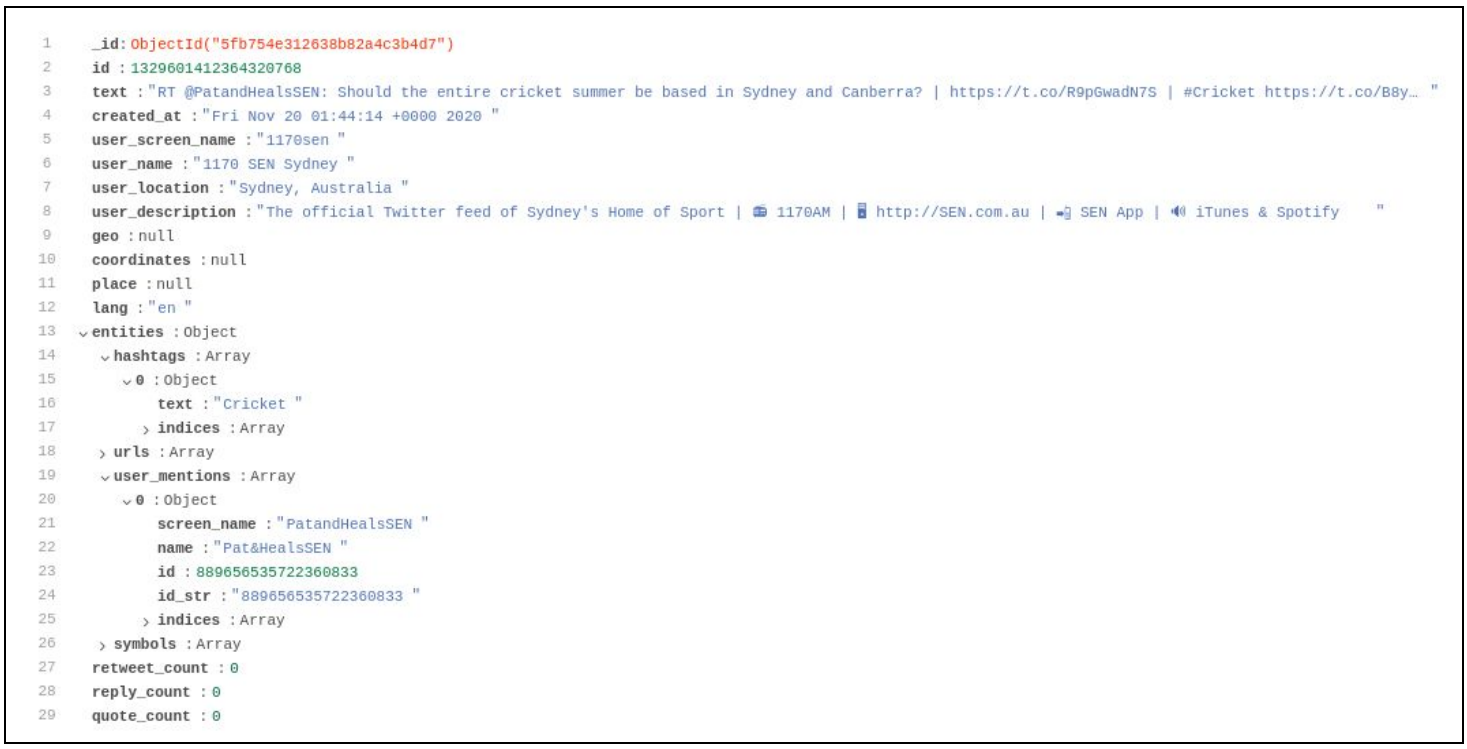
Data statistics:

After the data collection pipeline, we have around 150 thousand tweet documents in our MongoDB Twitter Collection. After building TF-IDF indexes and storing the weights, we have about 50 thousand documents in our MongoDB TFIDF Collection, which defines the number of distinct terms found in all the tweets we collected.

DB Schema:

We have one Database called twitterdb which contains two collections:

1. Twitter_Collection: this collection has tweet documents and the schema is as follows:



2. TFIDF-Collection: this collection has word documents containing the tfidf scores of each word and the lists of documents the word is present in. The schema is as follows:

Indexes:

1. TF-IDF - In information retrieval, tf–idf, short for term frequency–inverse document frequency, is a numerical statistic that is intended to reflect how important a word is to a document in a collection or corpus. It is often used as a weighting factor in searches of information retrieval, text mining, and user modeling. [source - Wikipedia]

$$w_{ij} = tf_{ij} idf_i = tf_{ij} \log_2(N / df_i)$$

The above formula is used for calculating the weight of each ith word in Jth document.

2. MongoDB indexes
   Indexing, by definition, allows us to reduce the number of documents the database needs to scan through to look for the specific document, thereby improving the performance of the database by leaps and bounds. To avoid MongoDB from performing COLLSCAN (collection scan), we will be using indexes. MongoDB offers several different types of indexes that we considered to build our search engine. Some of them are as follows:
   - Single field index: We considered creating single field indexes for each of the field that was being queried for, that is, 4 different single field index on user name, hashtags, location, and user mentions.
   - Compound field index: We also considered using a compound index that would contain all the fields that are queried upon. A compound index is an index supported by MongoDB where multiple fields can be indexed.

4. Tools used

The list of tools or libraries used:
1. Tweepy - A python library for accessing twitter API
2. Pymongo - A python library for connecting and interacting with Mongo database
3. Flask - Python framework used for backend APIs
4. Reactjs - Frontend Javascript framework
5. Tweet-Preprocessor - A python library for cleaning, tokenizing and parsing
6. Sklearn - Python Library for machine learning
7. Pandas - Data analysis and manipulation tool

5. Features

Data Engineering :

For developing a search engine the first and foremost thing is the requirement of a huge dataset. We here use twitter developer portal to collect tweets for a particular topic. We choose some prominent sport topics especially in cricket, football, tennis and formula 1. We develop a pipeline that collects data from twitter based on the above topics and puts the relevant fields in MongoDb.
As required in a search engine, we provide here a GUI to search based on text as well as on fields.

Text Search - A user simply can put some texts to search, the app would return the top five hundred relevant searches.

Once the data is collected, we retrieve the tweets and perform preprocessing. Some prominent steps are -

1. Removal of emojis, URLs, hashtags. This is done using Twitter preprocessor
2. Removal of punctuations, mentions etc.
3. Removal of stopwords like a, the, for get etc.

After cleaning and removing irrelevant information from the data we try to calculate a weight for each of the words in the tweet. This is done using TF-IDF as described above. So finally we have a matrix with words as the rows and tweets as columns. But the size of the matrix is large enough to be stored in-memory. So we create a new collection in MongoDb where we store the word score information. An object in MongoDb looks like


```
{
   "_id": {
      "$oid": "5fbca365fef0ffabc3187af6"
   },
   "word": "cricket",
   "docs": [{
      "docId": "5fb94e095610d30e4e222cda",
      "score": 0.49512879029226503
   }]
}
```


Here we can see the word “cricket” has docs and associated scores with it. We create a similar structure of the words we get and store them in MongoDB.

Whenever a user sends a query request of a few texts we fetch the relevant word objects from mongoDb. We calculate the score of each document by adding up the words and score present in each document.

For example, we query Ronaldo Real madrid:

	tweet1	tweet2	tweet3	tweet4
Ronaldo	0	0.1	0.2	0.0
Real	0.3	0.4	0	0
Madrid	0.1	0.3	0.2	0.5

Score Calculation:

Tweet 1 - 0.3 + 0.1 = 0.4
Tweet 2 -0.1 + 0.4 + 0.3 = 0.8
Tweet 3 - 0.2 + 0.2 = 0.4
Tweet 4 - 0.5

We see the score here is arranged in descending order is Tweet 2, Tweet 4, and followed by the rest,
Once the score is calculated we sort the tweets in descending order and send the top 500 tweets

Search based on fields:

To enable search based on fields of tweet document, we first decided on the particular fields we would allow being filtered on, which is as follows:

- user_name: The user name of the user who tweeted that tweet, this filter would output all the tweets tweeted by that particular user_name
- user_location: User location, which would output all the tweets tweeted from users from that particular location.
- hashtag: hashtag present in a tweet, this filter would output all the tweets containing that particular hashtag.
- user_mention: user mentioned or tagged in a tweet, this filter would output all the tweets in which the given user was mentioned

Once we had decided on the query that would be allowed, we considered the indexes we should use to facilitate high performance on this query. A typical request object to perform filtering on the above-mentioned fields looks like the following:

```
Request Object:
   {
       "user_name": "",
       "user_location": "",
       "user_mention":"",
       "hashtag": "",
   }
```

We had two options to perform a query on these fields, one would be performing an *AND* query which would return tweets that met all the given criteria and the other would be an *OR* query which would return the union of all tweets that met any or all of the given criteria.
To show the most relevant tweets at the very top, we first performed an AND query and then an OR query if the tweets returned from the first query were less than *500*. 500 here is the MAX value we selected as only the top 500 most relevant tweets would be shown or returned from our search on field service.

The most crucial part of performing queries was to create an index that would yield the highest performance on our queries. After a lot of research, we decided to use a Compound index.
MongoDB supports compound indexes, where a single index structure holds references to multiple fields within a collection's documents.
So we created an index that contained the fields: user_name, user_location, and entities.hashtags.text. One problem we encountered was when we tried to add entities.user_mentions.name in the compound query as this was a parallel array along with entities.hashtags.text, so MongoDB didn't allow it to be added to the index.
But a solution to this problem is to create a single field index on entities.user_mentions.name that would take care when the user_mention field is queried upon.
Even if a query containing all 4 fields happens, MongoDB has a functionality of index intersection which is inbuilt.
With index intersection, MongoDB can use an intersection of either the entire index or the index prefix. An index prefix is a subset of a compound index, consisting of one or more keys starting from the beginning of the index.
Due to this inbuilt feature, the compound index along with the single field index performs well and fulfills the queries required for the filtering functionality.

One other thing we had to keep in mind was the order of the index in the compound index created. If there is a need to query on only one key sometimes and at other times query on that key combined with a second key, then creating a compound index is more efficient than creating a single-key index but the order here matters as the index should have the fields that are most queried for in the beginning and the rest after that, as MongoDB uses index prefix intersections to recognize which queries will use which index intersections.

Indexes:


```
[
  {
        "v" : 2,
        "key" : {
                "_id" : 1
        },
        "name" : "_id_",
        "ns" : "twitterdb.twitter_collection"
  },
  {
        "v" : 2,
        "key" : {
                "entities.user_mentions.name" : 1
        },
        "name" : "user_mention",
        "ns" : "twitterdb.twitter_collection",
        "background" : false
  },
  {
        "v" : 2,
        "key" : {
                "entities.hashtags.text" : 1,
                "user_name" : 1,
                "user_location" : 1
        },
        "name" : "covered_index",
        "ns" : "twitterdb.twitter_collection",
        "background" : false
  }
]
```

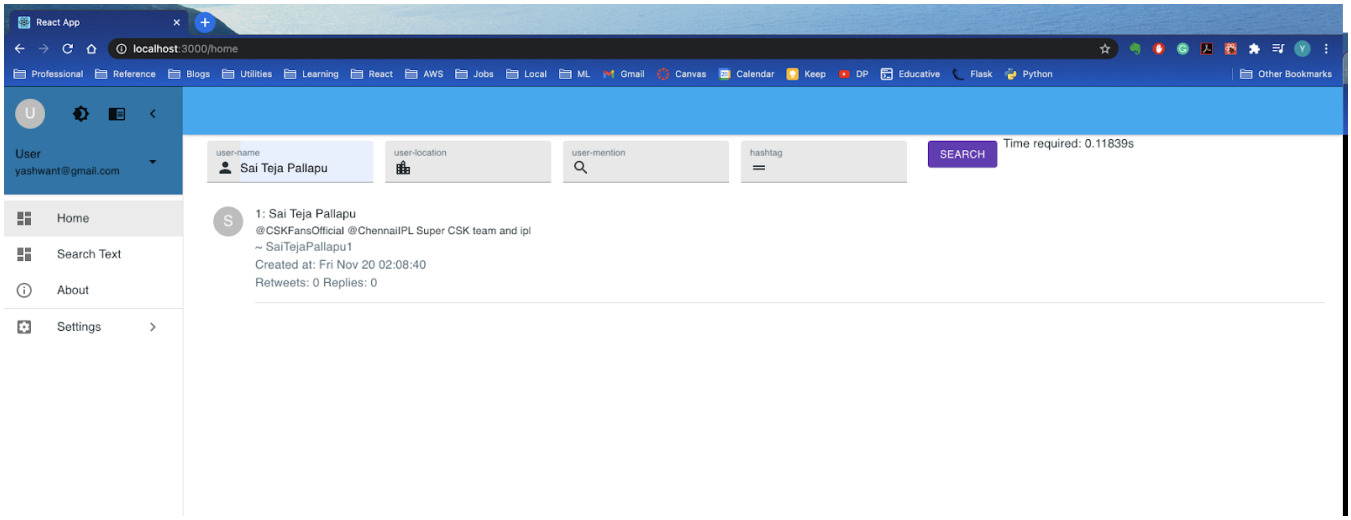

Queries performed:

- AND Query:
  {'$and': [{'user_name': 'Corporation Kev'}, {'user_location': 'USA'}, {'entities.user_mentions.name': 'SportsCenter'}, {'entities.hashtags.text': 'cricket'}]}

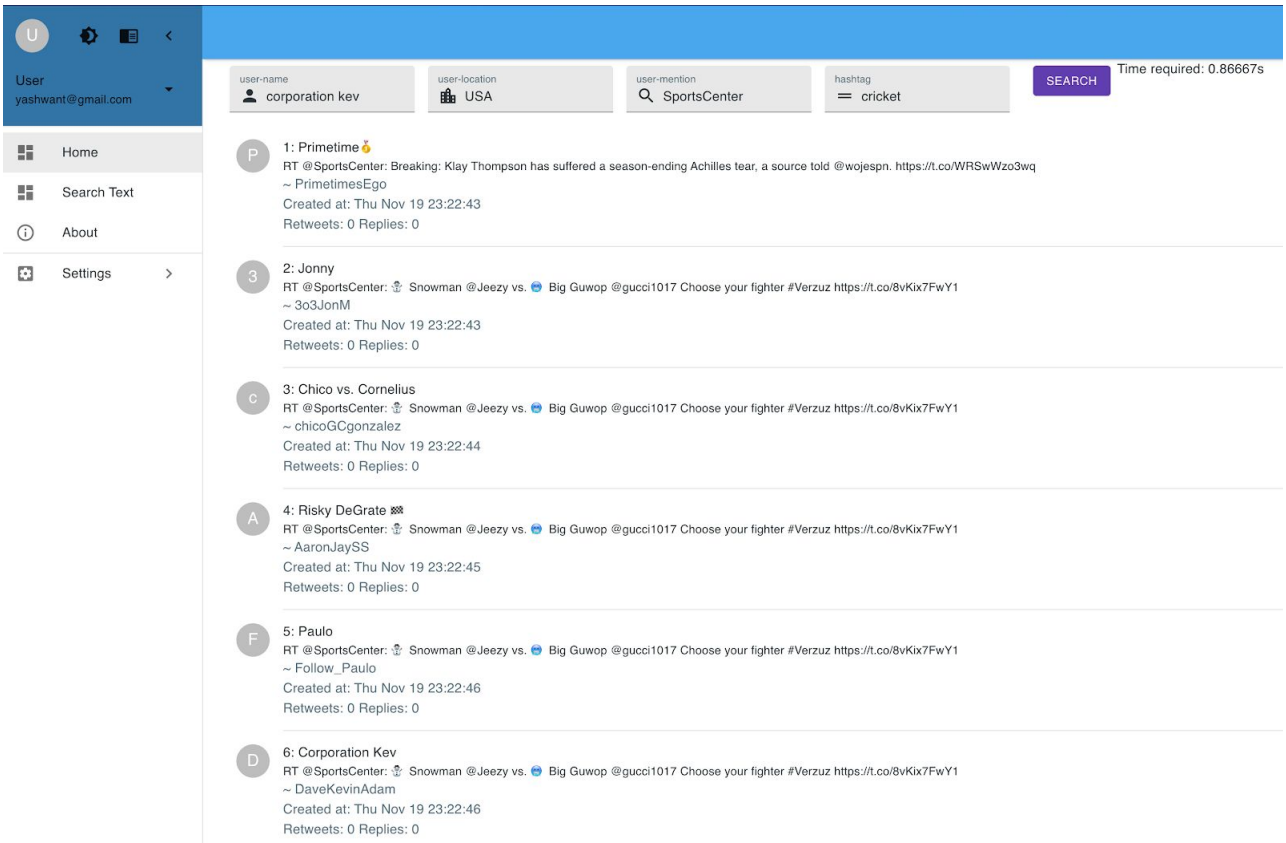
- OR Query:
  {'$or': [{'user_name': 'Corporation Kev'}, {'user_location': 'USA'}, {'entities.user_mentions.name': 'SportsCenter'}, {'entities.hashtags.text': 'cricket'}]}

Screenshots:

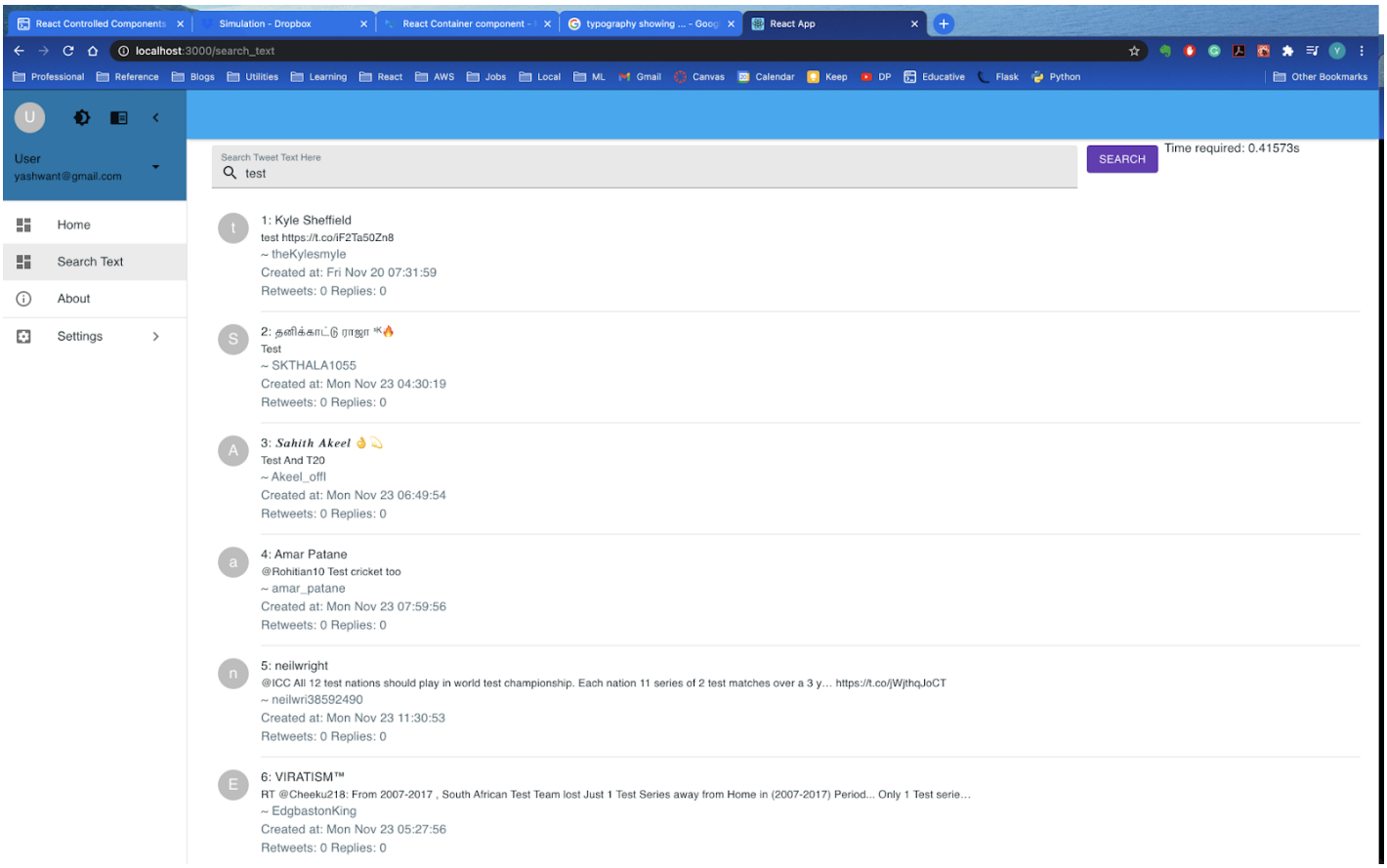
Search by fields



Search with multiple fields-



Search by text-



Lessons Learned

1. Handling Big Data- Initially, we tried to keep a tf-idf matrix in memory and query the matrix to get a result. Since the matrix size was near around 100k * 200k it took a lot of time to fetch it. We then used MongDb where in stored the values of matrix

2. Collection process could have been streamlined - We were dependent upon a third party for collection of tweets. It took a lot of time.

Open Issues

1. Collection process - We wrote scripts to scrape tweets from twitter developer API and to store it in MongoDB database, even then there was quite a lot of manual work involved in this process. We would want to make this pipeline more efficient and make it handle any exceptions thrown by twitter API.

2. Pre-processing text - In our preprocessing step, we cleaned the document of unnecessary punctuations, spaces as well as stop words, after which we tokenized the tweet text. We could've also used stemming and lemmatization to get more relevant results that might not exactly match the word but would match the meaning of the query being searched for.

3. Handling retweets - A twitter search engine is dependent upon tweets and the way they are being stored. Some of the tweets are retweeted and twitter just adds it as one more tweet. Basically more than one tweet exists with a different Id. We need to remove them as they are redundant. Removing them is a challenge as we need to match the text. Currently we are considering them as different tweets.

Future Work

1. **Database size** - The current database size is around 150k tweets in total which is scraped from the twitter public API.The Database size can be increased substantially to have analytics and other metrics to compare the performance.
2. **Use Elastic search** - To improve the search performance and easily scale the search system, Elasticsearch can be used.It also supports full text search and fuzzy searching which can help improve the user experience. It also supports multi-field search which can further improve the accuracy of the search results based on the context of the search query.

System Requirements (For linux based systems)

There are 3 important components for this application to run

1. Database - MongoDB
2. Python - Flask (Backend APIs)
3. Node - ReactJS (Frontend App)

Steps to Collect and Index data:

1. Enter Twitter Developer API credentials in the given strings in *scripts/stream.py* script
2. Run stream.py to create an intermediate JSON file in the same directory
3. Enter MongoDB connection string in the given string in *scripts/store.py (mongodb+srv://cmpe297-user:cmpe297-user@project-cmpe297.2ylzz.mongodb.net/twitterdb?retryWrites=true&w=majority)*

4. Run store.py to relay all the tweets from JSON file to MongoDB twitter_collection.
5. Run search.py to create TF-IDF indexing of tweets from twitter_collection and store the tf-idf scores in the respective collection in MongoDB.

Steps to run the application:

1. For the backend to start running as a service use the terminal and api folder:
    a. Install the requirements for all the libraries to work using the following command
        pip3 install -r requirements.txt
    b. Start the flask application
        flask run
2. For the frontend navigate to the api folder:
    a. Install all the libraries to work using the following command
        npm install
    b. Start the flask application
        npm start

Github

https://github.com/purvimisal/Twitter-Search-Engine