CS242 Project Report Part B

Kuai Yu, I-Hsien Huang, Yifei Lai

A. Collaboration Details

All three group members participated in the production of this report and shared contribution in the design and implementation of all systems. However each member is also responsible for a particular phase / implementation within the project with the following distribution of responsibility:

I-Hsien Huang - Part A (Twitter4j Implementation + Data Parsing) +
Part B (NoSQL Cassandra database, document ranking calculation)
Yifei Lai - Part A (Lucene Implementation + Indexing Strategy) +
Part B (MapReduce Implementation + Removing stop words and indexing)
Kuai Yu - Part A (Tweepy Implementation + Raw Data Coalescing) +
Part B (Word Stemming + Front-end Integration)

B. System Overview

Architecture

Our system has three main components which are the indexing system, database system and front-end web system. For the indexing system, we have two different ways to index data: MapReduce of Hadoop and Lucene. BEcause to the output indices of Lucene is in a well-organized form, there is no need to put them into the database. Thus, we only combined the indices of MapReduce with the database system. We used the NoSQL database Cassandra to store our indices and data. It contains two tables, one is for the original data and the other is for the indices. We used the query words to search the index table first, and then using the results which are the information of the words to search the table containing the original data. The document ranking algorithm such as BM25 was processed in this database system too. The front-end web system is built using PHP, jQuery and Bootstrap, and we used the Apache HTTP web server to host these files. All interaction between the webserver and the backend database system are done using calls to Python functions.

Hadoop

We mainly used the MapReduce function of Hadoop. It is a good way to gather all the values of the same key together for further usage. The keys here are the words of the texts or locations of the tweets, and the values are the document IDs, positions of the words in a tweet and so on. In our project, we used one of our computers to implement MapReduce. The reasons are that the

computer has installed and deployed Hadoop before, and we could use some IDEs such as Eclipse to help with coding and debugging.

Our indexing procedure has one MapReduce process. For the map phase, we have two steps: The first one is to analyze the input file which is in a JSON format. We used the same method as the one we used to analyze the Lucene input. Note that each line is a tweet in the input file and each tweet is a JSON string. The line number is the same as the document number. In this step, we extract the text and location segments in the JSON string. In the second step, the text segments and location segments are analyzed by two MapReduce processes, but the method is the same. Let's take the text segments as an example. Each word in the text segment will be checked to see whether it is a stop word and set as the key for mapping if it is not. The line number will be recorded if this line contains the word, and the position of the word in this line will be combined with the line number in the form of "Line Number:Position". These index information will be set as the value of its corresponding word.

In the reduce phase, we built a hashmap. For each unique key, we get its value iteratively, and each value will be split by the ":" delimiter, the left value is set to the key of hashmap, and an integer is its corresponding value. Each time we get the same line number which is a document ID, we will add one to its value. As a result, we can get the frequencies of occurrence of this word in different documents. We also have an integer to record the total frequency of this word in the whole collection. Finally, we combine the total frequency, document frequency, and index together as the value. They are presented in the following format:
"TF:TotalFrequency;DF:DocId1:DocFrequency1;DocId2:DocFrequency2;....;I:DocId1:Position2;DocId2:Position2…..;"
The key and this long value string are subsequently outputted to a file for further processing.

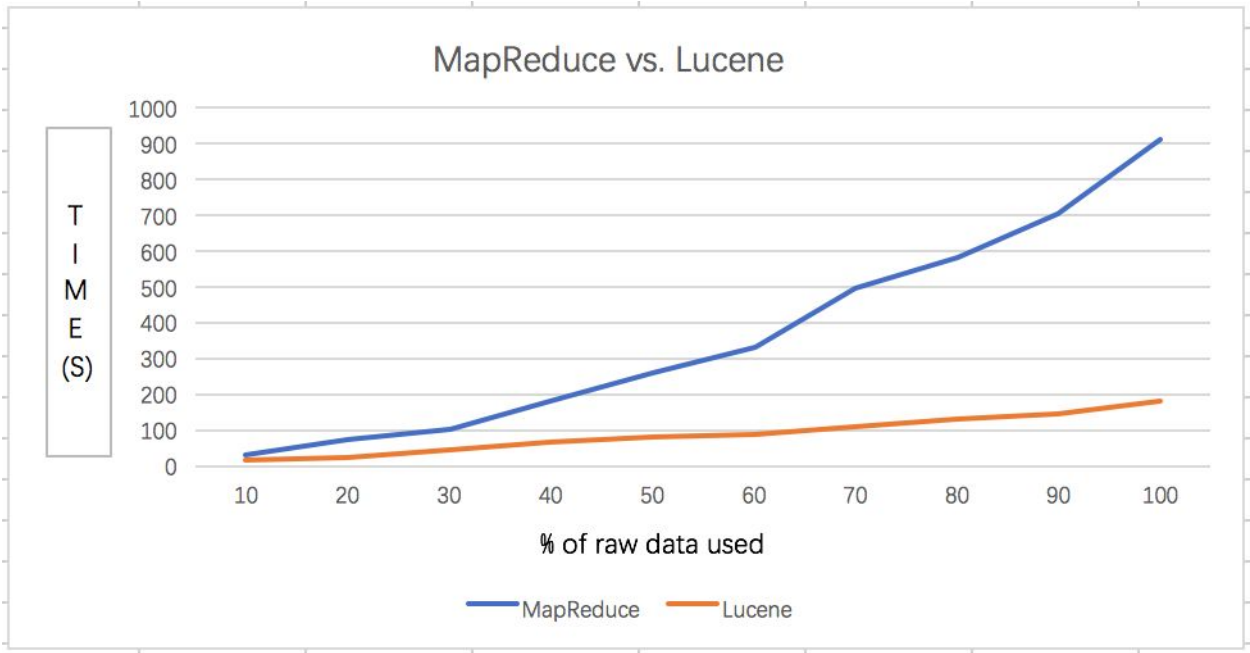
Lucene

Lucene is a well-organized system. It can not only build the index fast, but also do the query efficiently. Like we have discussed in the first project report, we firstly should extract useful information from the JSON string. Then use the StandardAnalyzer and IndexWriter objects to build the indices. The indices are stored on the disk.

In the query phase, we decided to rank the results by the "number of followers" value of the twitterer who published this tweet. At first, we use the IndexSearcher to search for the index and record the results of some given words. The results contained information such as the tweet content, published location, twitter IDs and so on. Then, the followers numbers of those results are set as the keys of a TreeMap, and the rest of the results information are set as the value corresponding to the key. Because TreeMap performs automatic ordering of the keys, after changing those values into the UTF-8 format, we are able to directly output the values in a descending order.

Runtime Comparison

By compare Lucene and Hadoop, we can find that Lucene is really efficient. It is much faster than Hadoop when built the index. However, nothing is perfect, and the time complexity to modify part of the inverted index on Lucene is much higher than Hadoop.



NOSQL: Cassandra

Cassandra is a kind of NOSQL that means the data structure is stored in a key-value from. Therefore, it is very efficient to search, with a time complexity of only O(1). Besides, compared to HDFS which has a single namenode, in Cassandra, there are multiple data centers with built-in redundancy. Consequently, we do not need to worry about handling the crash of one data center. Furthermore, Cassandra also supports SQL parsing. However, cassandra is not designed for big data, with a size of thousands Terabytes.

In our project, we utilized cassandra to store our inverted index and original data in two databases. The web application would send the query to database1 and get the doc_id of the words, and then send the doc_id to database_2. The database_2 would then calculate the document rank with several algorithms of the user's choosing, like BM25, popularity or BM25+ location and send back the tweet information, sorted by document rank.

Apache HTTP, PHP, Bootstrap & jQuery

In order to create a usable interface for our search engine we decided to use the Apache HTTP web server from the XAMPP library. One advantage of using XAMPP was that it was easy to set up, allowed for cross-platform installation and automatically came with a PHP interpreter. We also used jQuery to perform animations of our webpage combined with two Bootstrap templates

we modified to be our landing page and search page respectively (see bibliography for references). To bridge the user input with the search engine output, we used PHP to dynamically call various Python functions (which returned the query results in ranked order). PHP is then used to dynamically update the query result table for users to see (and also used to perform the matching-text highlighting feature). Stemming of initial user input is again performed by a PHP call to a stemming python function which we created.

Stemming

Word stemming was originally planned as a part of Hadoop. However owing to memory limitations of the machine with which we used to run Hadoop instances, we decided to move word stemming outside of map-reduce as a separate function. Instead of outputting stemmed words to file from Hadoop, we instead ran a highly efficient python parser we constructed on the inverted index to combine words (and their associated occurrence count, occurance per document count as well as location within each document) according to each word's root word. For example, inverted index entries under the terms "computer, computing, computers" would all fall under the same root word "comput". Note that because we used the Python PorterParser library in python (which exhibited some aggressive stemming behavior), root words are not necessarily english words. However this will not affect query results because all passed in queries will also undergo stemming by PorterParser, so only words with valid root stems will be matched. After parsing, a total of 19222 root words are created and stored in a hashmap compared with 1008677 total unique words in our inverted index dictionary. 972312 unique, non-stemmable words are also indexed. In the end, around 3.6% of our dictionary is stemmed. This shows that the majority of words from our twitter database are unique words or proper nouns.

C. System Limitations

Twitter Dataset Size & Historical Dataset

One problem which we did not forsee was the amount of garbage data we would collect using the Twitter Streaming API. Because we wanted to obtain tweets from every possible location (so we may use our BM25 + Location ranking algorithm), we managed to obtain a large number of tweets that were not written in English, or even ASCII. This meant we had to perform extensive trimming of the dataset during data preprocessing on non interprable words. This in turn meant that although our original dataset was of size 6.7G, the amount of usable data was less than this size. It turned out during testing of our system that certain rare words were sparse in our dataset. This is especially the case when we performed testing of our BM25 + Location ranking algorithm and narrowed down the results further using a location filter. Therefore for future system implementations, we recommend crawling at least 12G of raw data before creating the inverted index.

Word Stemming in Hadoop

Another limitation of our system is that owing to our implementation of our word stemming function in Python using the PorterStemmer library, we had to perform stemming on the inverted index itself and not during map-reduce. This meant that there is a time penalty for creating the stemmed index of around 5 minutes. We recommend that in future implementations that word stemming be moved to Hadoop and performed using a Java library so that stemming time may be negated. That being said, 5 minutes to stem and combine stemmed words in our inverted index is sufficiently quick for the amount of data we crawled.

D. Obstacles and Solutions

Handling Unicode Inputs

Although we filtered out much of the non-English tweets from our dataset, we still encountered many non-ASCII characters when we performed the indexing operation and subsequently during the output of query results to screen using PHP. This was resolved by making sure all query results returned by the three python functions (for calling BM25, Popularity and BM25 + Location ranking algorithms respectively) outputted results in Unicode instead. This allowed us to display not only non-English characters but also emojis, which were prevalent in tweets.

E. Deployment Instructions

We also provided a video demo in our submission folder to showcase our search engine.

1. Install the Cassandra system, download apache-cassandra-3.9.
2. Before run Cassandra, run this command line
   export PATH=$PATH/filepath/apache-cassandra-3.9/bin
   $Cassandra -> $cqlsh -> create keyspace -> create 3 tables (stemming database, map_reduce database, original database)
   Keyspace_name :info
3. There are two Lucene runnable jar file(Lucene1.jar, Lucene.jar)
   Lucene1.jar : call it to build inverted index
   java lucene.jar “query” : call it to search the query from inverted index.
   And we also submitted luceneindex.java, the original java file.
4. Python file introduction:
   Python Version: 2.7
   Data_build.py : store data to cassandra database1.
   Index_build.py: store stemmed inverted index to Cassandra database 2.
   Location_index.py: store the location inverted index, processed by Hadoop.
   ($python python_file_name)
   Three following python files are used to received query and return twitter information, and the only difference is the document algorithm.
   data_select.py: algorithm BM25

Data_select_followers.py: algorithm: calculate by the number of the user's followers.
Data_select_location.py: algorithm is BM25+ location limited
($python python_file_name "query")

5. Running the MapReduce:

Make sure the Hadoop is installed and started. If you use the IDE like Eclipse, make sure the libraries of Hadoop have been loaded into the IDE.

Make sure the json package is in the same src fold in the IDE or compiled together with the original java file. The data we crawled should be put into the input fold.

To build the index of tweet text:

IDE: In the properties of the indexing.java, set the parameters as "input output". Then running it.

To build the index of tweet location:

IDE: In the properties of the L_indexing.java, set the parameters as "input output". Then running it.

6. Running the web server:

First install an instance of the XAMPP program for your environment (link in bibliography).

Then perform the following commands:

$ cd /opt/lampp/htdocs

$ sudo mkdir cs242v2

$ sudo chmod 777 -R /opt/lampp/htdocs/cs242v2

In this directory, paste in all files from the directory cs242v2 in our submission folder.

To start the Apache HTTP Webserver, use the following command:

$ cd /opt/lampp && sudo ./xampp start

To restart the server, in the same directory, use the following command:

$ sudo ./xampp restart

To view the webpage, enter <localhost/cs242v2/index.html> into your web browser.

7. Ensure the previous step is correctly carried out and perform the following check:

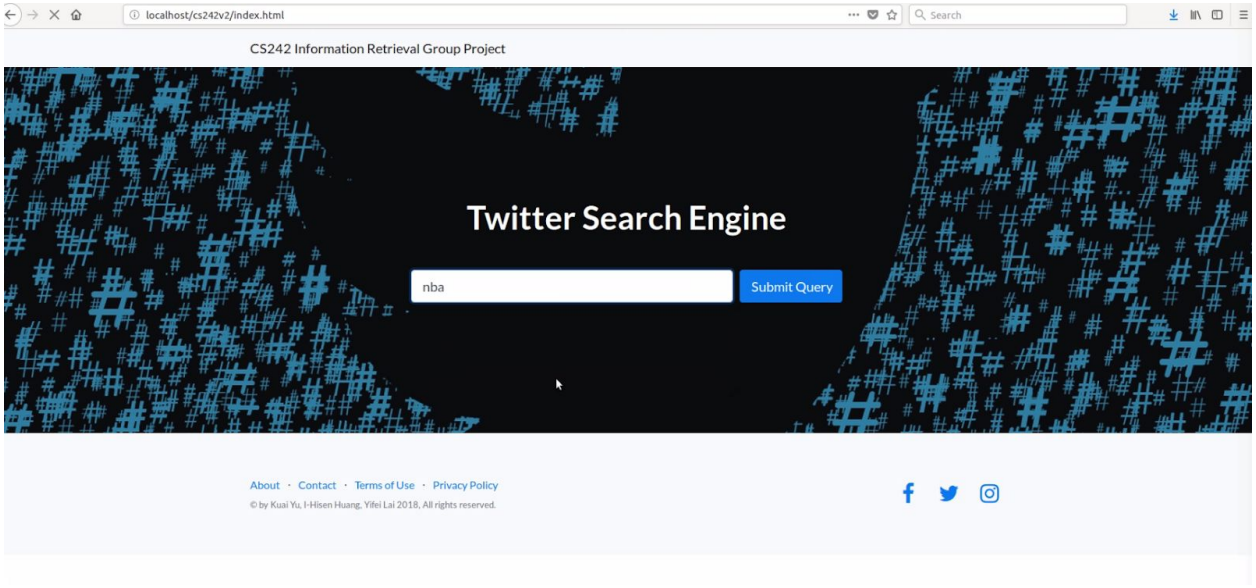
Go to the landing page and enter any query. If the resulting page does not return a query result, then it may be the case that cassandra is not properly installed in your XAMPP environment. Perform the following installation:

$ sudo pip install cassandra-driver

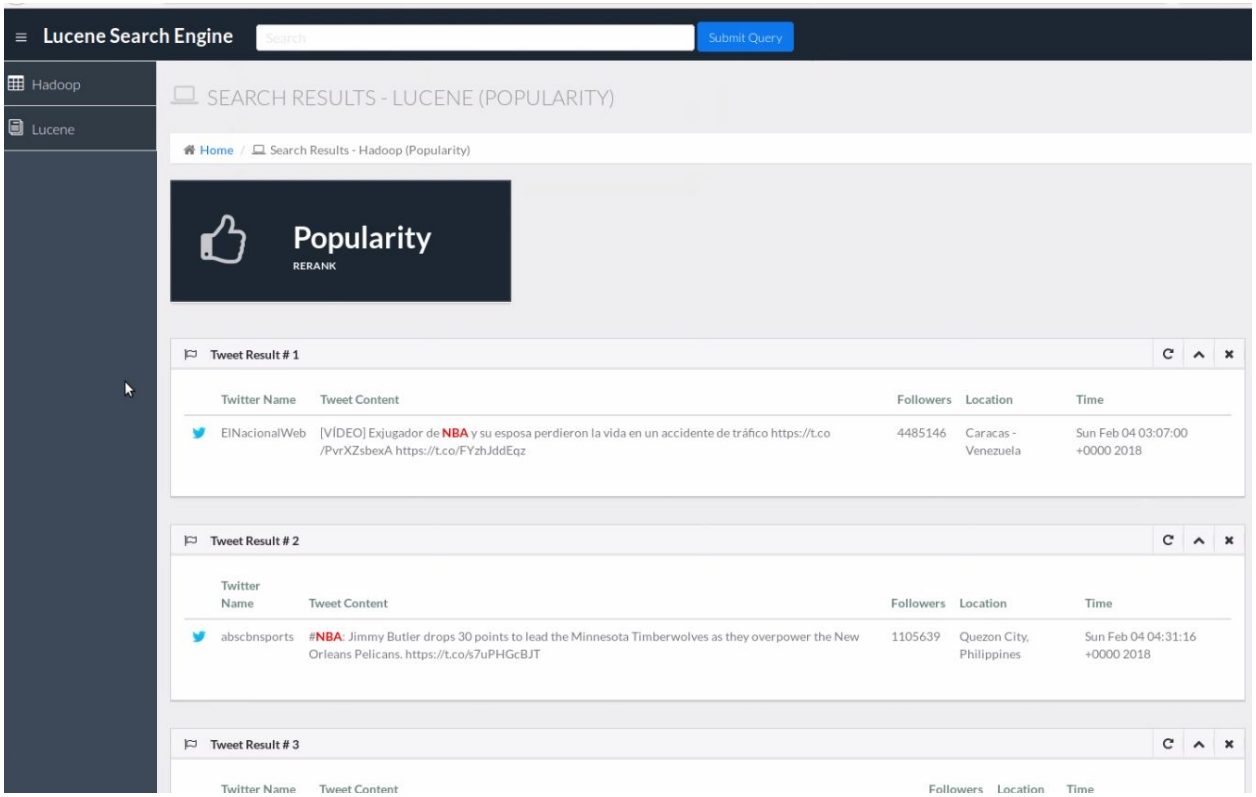
8. Finally if you experience any errors, please contact our group members.

F. Screenshots

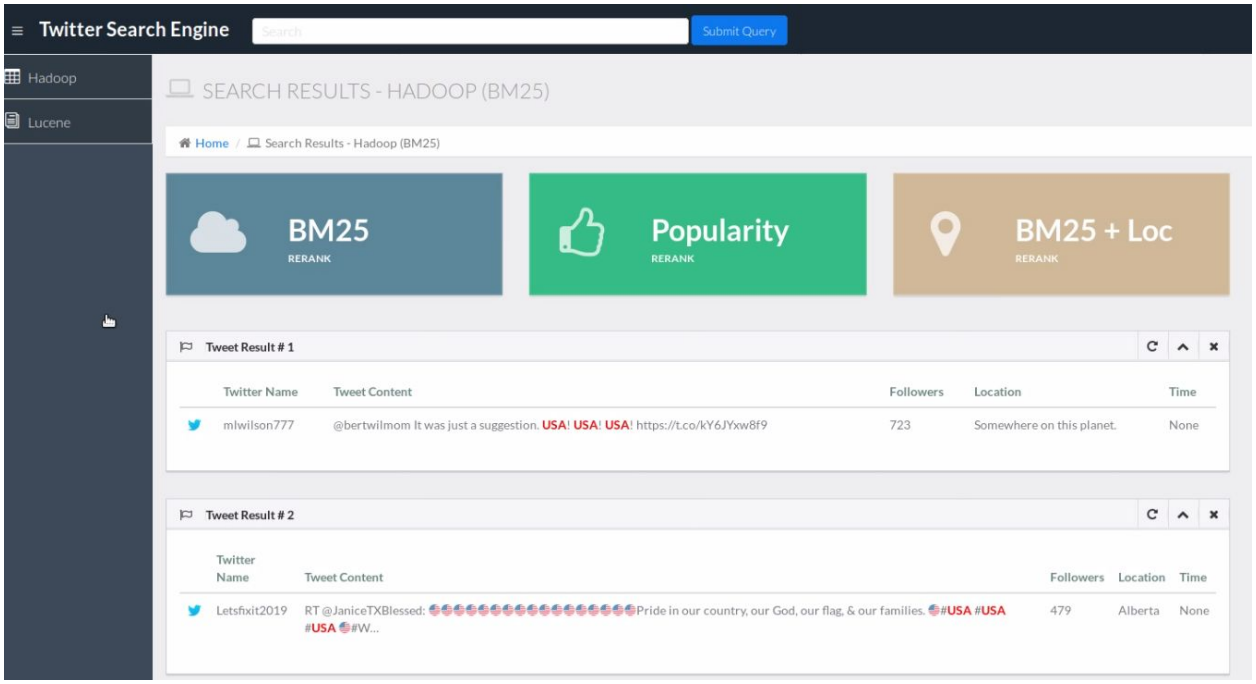
UI Introduction:



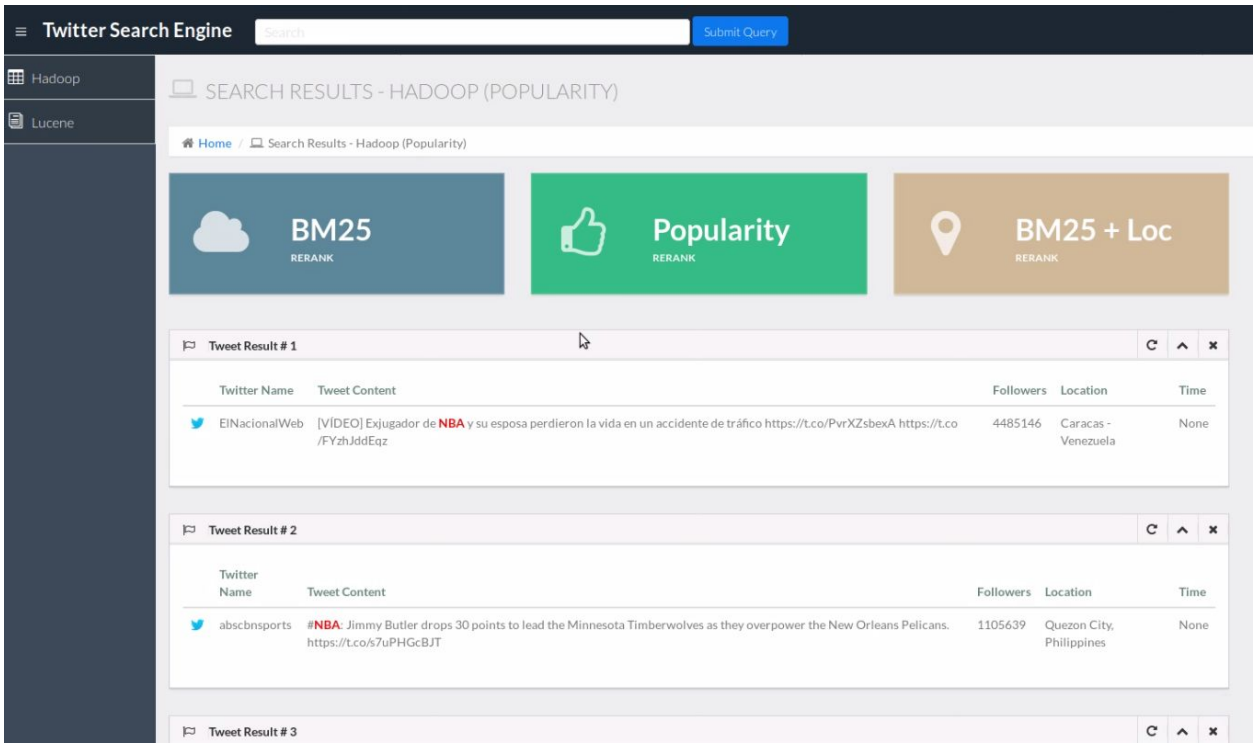
1. Lucene search result
   Insert: NBA



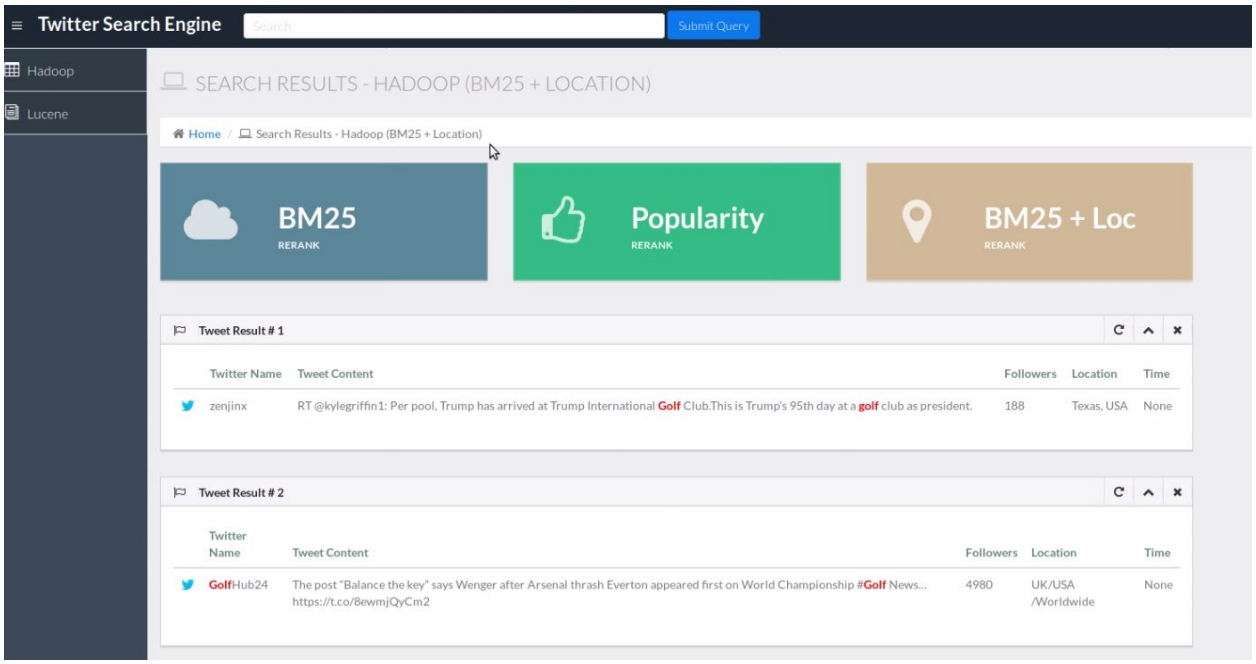
2. Hadoop BM25 algorithm search result
   Insert: USA



3. Hadoop popularity algorithm search result
   Insert: NBA the result should be the same as Lucene if and insert querys are the same （



4. Hadoop BM25+Location algorithm search result:
   insert : golf usa -> limit the twitter is released in the location usa

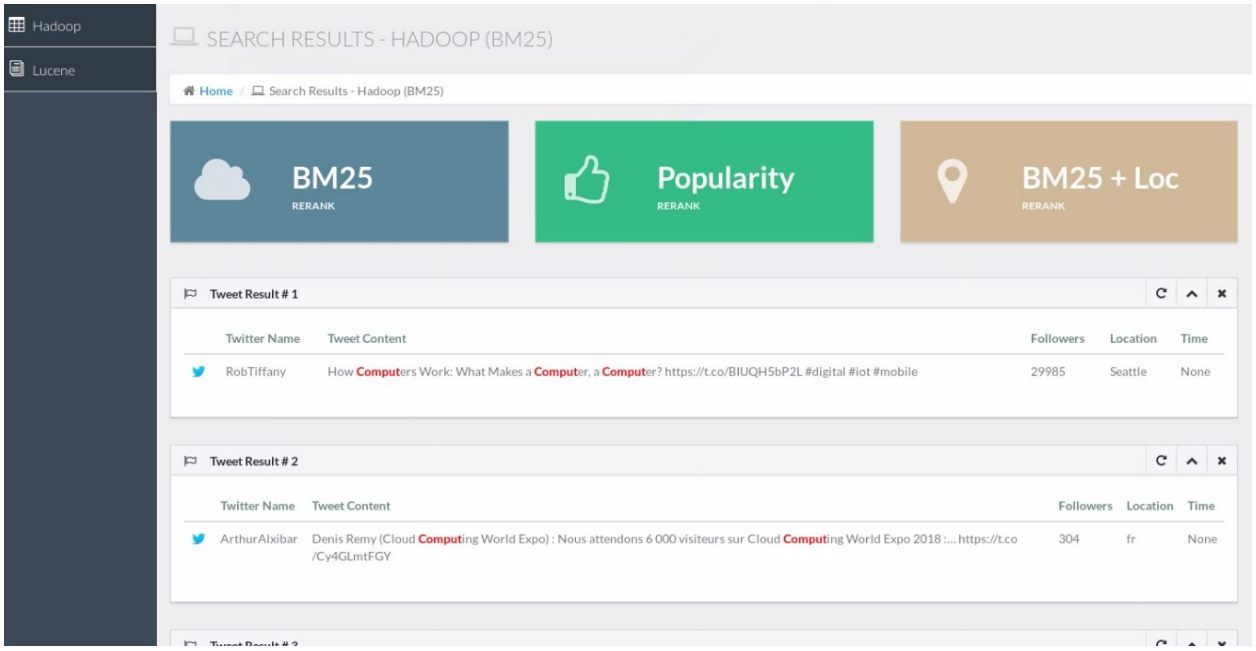


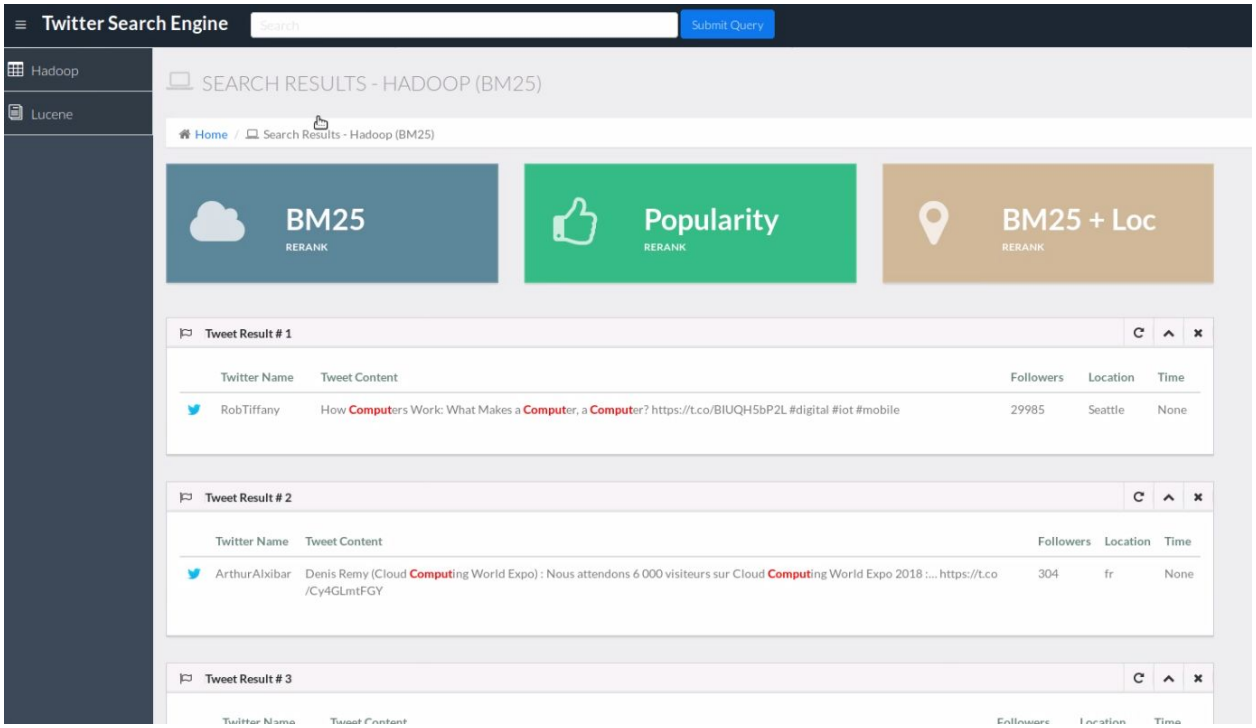
5. Hadoop stemming search result :

   Insert: computer

   Insert: computing

   The result are the same, because of the stemming function.





N.B. [ALL OF OUR CODE ARE PUT inside **cs242v2/ResultPage/**]

G. Works Cited

Loen. “Cassandra教程.” W3Cschool, 6 Jan. 2018, <www.w3cschool.cn/cassandra/>.

Changhau, Issac. Hadoop Installation on Mac OS X. 27 June 2017, <https://isaacchanghau.github.io/2017/06/27/Hadoop-Installation-on-Mac-OS-X/>.

xpo6. “List of English Stop Words.” XPO6, 14 Apr. 2009, <xpo6.com/list-of-english-stop-words/>.

“Lucene - StandardAnalyzer.” Tutorialspoint, <www.tutorialspoint.com/lucene/lucene_standardanalyzer.htm>.

“Landing Page.” Start Bootstrap, <startbootstrap.com/template-overviews/landing-page/>.

“Nice Admin -Free Bootstrap Admin HTML Template.” BootstrapMade, <https://Bootstrapmade.com/Nice-Admin-Bootstrap-Admin-Html-Template/>, 2018.

“Apache Friends.” Download XAMPP, <www.apachefriends.org/download.html>.