

CS 242 Project Report II

Tweepy Search Engine using Hadoop and Lucene

Group 8

Madhusudhan Tungamitta (862325125)

Thirumalai Vinjamoor Akhil Srinivas (862320790)

Balaji Arunachalam (862321425)

Shreya Godishala (862313765)

Imran Shahid (862319859)

Summary Details:

1. Collaboration Details
2. Why twitter
3. System Overview and Architecture
 - a. Project Architecture
 - b. Languages Used
 - c. Crawling
 - d. Lucene/Hadoop Indexing
 - e. Web application
 - f. Search Engine Result
 - g. Back End(Lucene index to Return results.)
4. Hadoop Usage key values definition and data flow
5. Hadoop Runtime Index Construction
6. Hadoop created an Index to ranking
7. Hadoop Runtime Vs Lucene Runtime
8. Limitations of systems
9. Obstacles and Solution
10. Instruction on deploying the system
11. Indexers file

1. Collaboration Details:

We are a group of 5 people working on this project. For the first phase of the project, two-part of the project are split into 2 people for Twitter Crawler and 3 people for Lucene. For the second phase 2 people worked for Hadoop Indexing, 3 worked for Front end connection.

Name	Contribution
Madhusudhan Tungamitta	Worked on Twitter crawler and UI design and connection, the idea of using geolocation and implementation
Balaji Arunachalam	Worked on Twitter crawler, Hadoop indexing, Coding and determining structure, Debugging
Shreya Godishala	Lucene index creation and quality testing with large datasets. Crawling data, Hadoop ranking, System and Environment setup
Imran Shahid	Lucene data creation, architecture design, spring boot backend connection, Hadoop Indexing
Thirumalai Vinjamoor Akhil Srinivas	connecting backend to front end, monitoring and obtaining data, Lucene Searcher class, Debugging and file structure

2. Why Twitter:

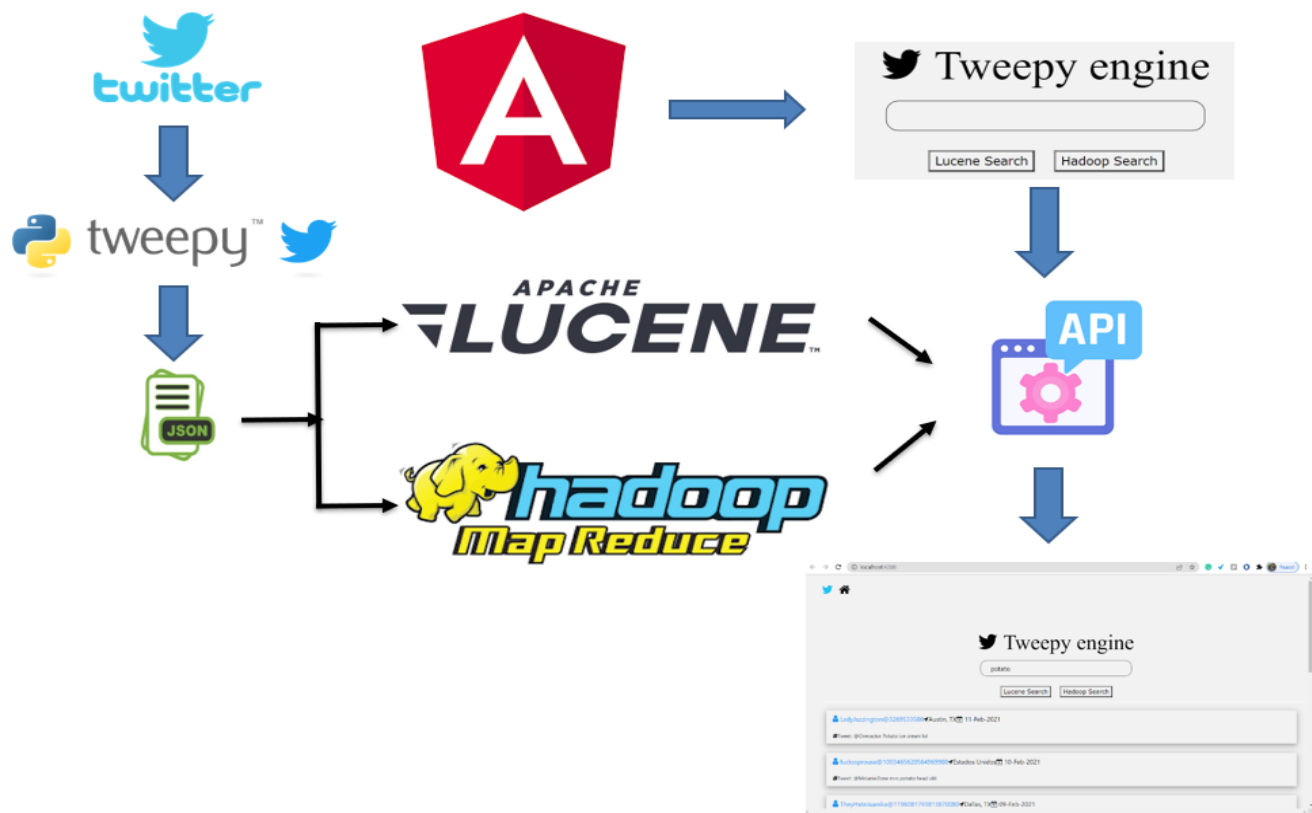
Twitter is an excellent source of data. It gives us the freedom to crawl the data with the official API they are providing. We are using Tweepy in this project for crawling the data. The API class provides access to the entire Twitter RESTful API method. Each method can accept various parameters and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

3. Overview of the system:

1. Crawling(Tweepy)
2. Lucene/Hadoop Indexing
3. User Interface

3.1 Project Architecture:



3.2 Languages Used:

Languages used: Python3, Java, Spring boot Framework, Typescript, Angular Framework

Api: Tweepy API

Cloudfare: For designing the fonts and icons to display

3.3 Crawling:

We are basically crawling based on the topic which is given as an input (ie., ICC or cricket). It will crawl tweets based on that query.

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

Config.py:

Config file contains the API keys/consumer key, API/Consumer Secret, Access key, Access key secret is generated from the app that we have created in Twitter. For this, we created the app in the Twitter developer account.

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.

```
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 also written a separate python file twittercrawler1.py which pulls all the tweets with geolocation tagged and non-geolocation tagged tweets and stores them in two separate files.

3.4 Lucene/Hadoop Indexing

3.4.1. Lucene Indexing:

The design of the Lucene indexer is as follows: It is a 3 part system composed of a parser, an indexer, and a searcher class. The indexer class utilizes the `indexWriter()` and `StandardAnalyzer()` methods to create indexes and write to them.

```
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){

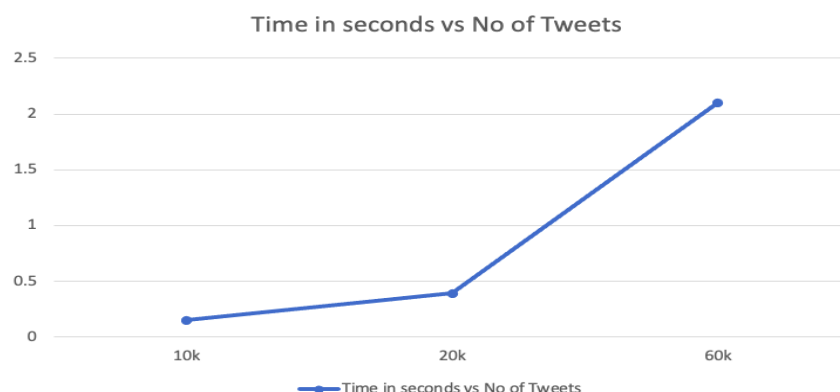
        String indexWriter = "timestamp";
        Field timestamp = new StringField(indexWriter,tweetParserObj.getTimeStamp(),
Field.Store.YES);
        doc.add(timestamp);
        this.addToIndexFieldList(indexWriter);
    }

    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);
    }
}
```

The parser class is called each time a tweet is indexed as it is needed to break down the tweet into its component fields.

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[40 mb].



3.4.2 Apache Hadoop

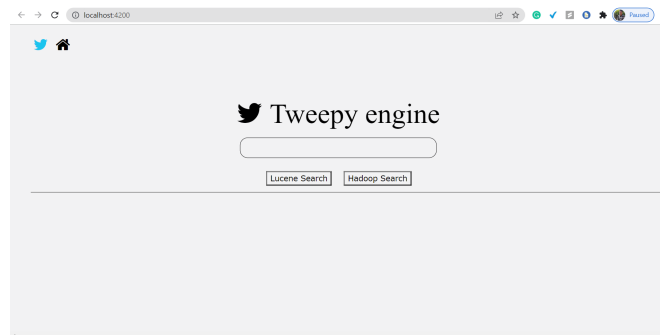
The Apache Hadoop is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models.

- HDFS, YARN, and MapReduce are the three main components of Hadoop.
- HDFS is a distributed file system that provides high-throughput access to application data.
- YARN is responsible for job scheduling and cluster resource management.

MapReduce is used to process a large data set parallelly by splitting the task up into smaller MapReduce Jobs.

3.5 Web Application (Angular + spring boot):

Our web application consists of a simple search box and two buttons(Lucene, Hadoop). If we click on Lucene search Result It will retrieve Lucene indexed data and when we click on the Hadoop button it will retrieve the Hadoop indexed data and display it in the UI.



Basically, our web interface looks like the above image, we have used angular for the front end including HTML, CSS, typescript, font awesome in the front end part. Using Ngif condition is provided an attribute of the anchor element for the inserted grid template and angular expands into a more explicit version in which the anchor element is contained, using that it will expand the grids and display the top 10 tweets. We are retrieving the location, user name, tweets,

When we click on the Lucene/Hadoop button it will call the tweet service typescript file. In the tweet services file, we have written an HTTP request and it represents the outgoing request like the URL method, in our case, it represents the localhost 8080 which is the port that is used to run our spring boot application.

```
private baseUrl = 'http://localhost:8080';

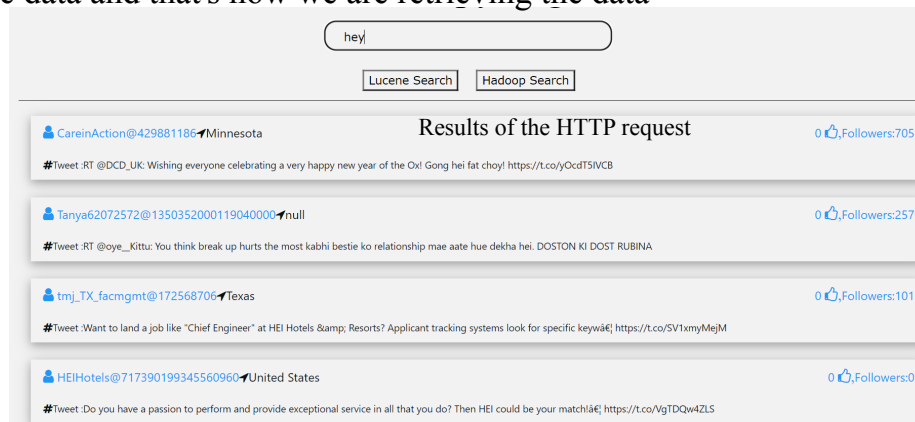
constructor(private http: HttpClient) { }

getTweets(query: string): Observable<Tweet[]> {
  return this.http.get<Tweet[]>(this.baseUrl + '/api/tweets?query=' + query);
}
```

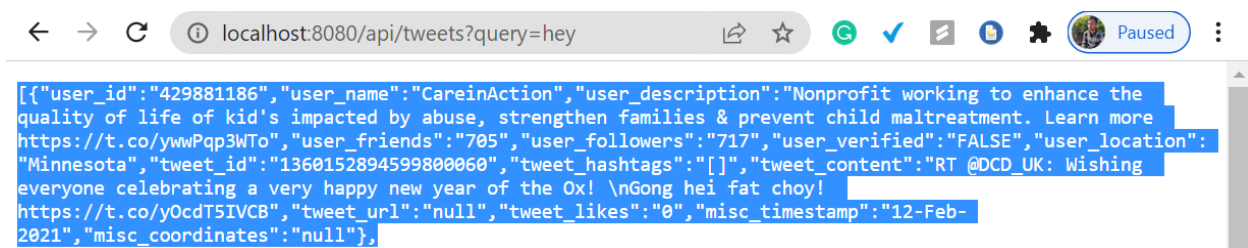
HTTP requesting angular

3.5.1 Search Engine Results:

We are getting the results from the spring boot and when we click on the query also we can see the data and that's how we are retrieving the data



3.5.2 Back End(Using Lucene Index to Return the results):



3.5.2.1 Reading data

Indexed output data from the Lucene and Hadoop is stored in two folders user index and tweet index folders and we are setting the indexed data directory to access the data and using the directory reader we are reading the data.

```
Directory userDirectory = FSDirectory.open(Paths.get("user_index"));
Directory tweetDirectory = FSDirectory.open(Paths.get("tweet_index"));
```

We used the IndexReader() and MultiFieldQueryParser() to open and parse the index created by Lucene and then we specify the number of fields to return results these fields are as follows: user_id, tweer_id, hashtags, and content. The same user features are taken as well.

```
DirectoryReader userIndexReader = DirectoryReader.open(userDirectory);  
DirectoryReader tweetIndexReader = DirectoryReader.open(tweetDirectory);
```

3.5.2.2 Getting tweet and user data

In for loop get the tweet data based on the ranks to get the tweet data and then using the tweetdoc we can use this to get the user details like user id, username, user description, friends, followers, location, tweetid, likes

```
Document tweetDoc = tweetIndexSearcher.doc(scores[rank].doc);
```

3.5.2.3 Index Searcher:

We then specify BM25 similarity with the IndexSearcher() method and also specify weights that the searcher uses to balance results.

```
userIndexSearcher.setSimilarity(new BooleanSimilarity());  
tweetIndexSearcher.setSimilarity(new BM25Similarity());
```

We then pass the data through an EnglishAnalyzer method and a Keyword analyzer method. We then return the score by using the inbuilt method and then print by the highest score after the query. As the fields are weighted in order we can immediately use the parser and index reader to return a weighted score that is representative of the tweet and the input query.

3.5.2.4 Storing the results

We are appending the results and returning the results and using this results value it will get the user and tweets details and store it wherever we gave the query that is matching


```
Tweet result = new Tweet(user_id, user_name, user_description, user_friends, user_followers, user_verified, user_location, tweet_id, twe
results.add(result);
```

When we run the spring boot application all these field values will be returned and displayed in the json format when we give the localhost:8080/api/query in web browser

3.5.2.5 Weights:

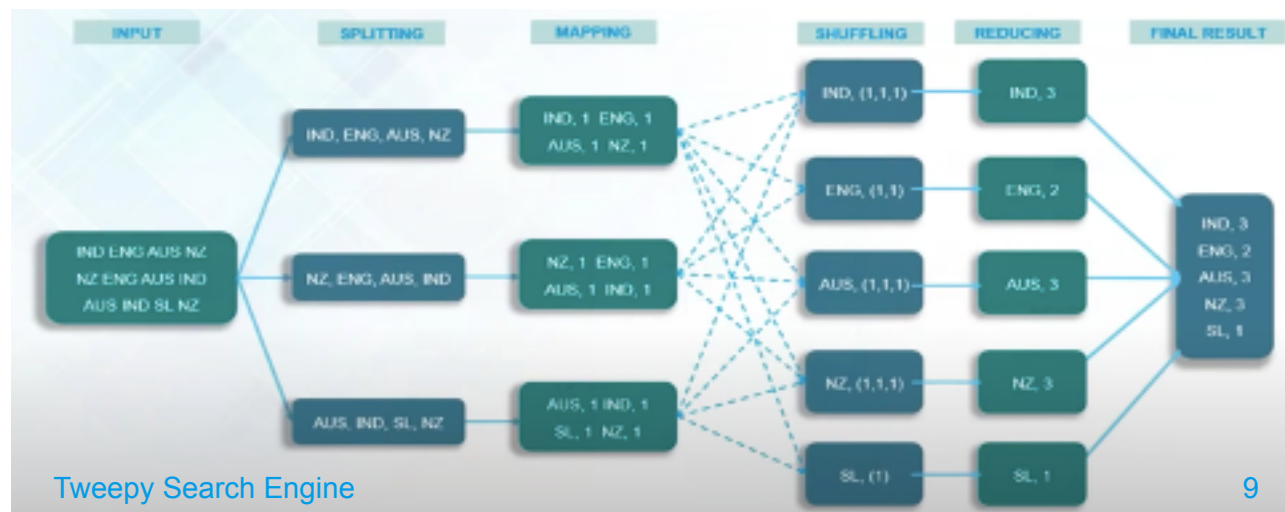
The searcher class makes use of two methods to identify and score the input query and the tweets associated with it. We access the indexes and then we use the IndexSearcher along with the MultiFieldQueryParser() methods to weigh the various fields being parsed and to also set the similarity criterion for the searcher.

```
Map<String, Float> tweet_weights = new HashMap<>();
tweet_weights.put(tweet_fields[0], 1.0f);
tweet_weights.put(tweet_fields[1], 1.0f);
tweet_weights.put(tweet_fields[2], 1.0f);
tweet_weights.put(tweet_fields[3], 1.0f);
tweet_weights.put(tweet_fields[4], 1.0f);
tweet_weights.put(tweet_fields[5], 1.0f);
tweet_weights.put(tweet_fields[6], 1.0f);
tweet_weights.put(tweet_fields[7], 1.0f);
```

In this case, it is BM25 and Boolean similarity, We then score the tweets returned by the query and return them.

4. Hadoop Usage key, values definitions, and data flow.

The tweets are pulled from Twitter based on a query with help of Tweepy, an easy-to-use library that python provides to communicate with the Twitter API and retrieve tweets. All these tweets have been stored in a document that must be indexed using Hadoop MapReduce. In MapReduce, the process takes place in five different phases and consists of 3 main classes: Mapper, Reducer, and Driver. This Driver is used to communicate the Mapper with the Reducer class and scheduling and the resource management will take place in it. In Mapper class Splitting and the Mapping phase will take place which pulls the tweets from the document we stored line by line, in the Splitting phase once splitting is done, Mapping takes place and gets the tweets, tweet count, and the document Id. These will be then sent to the Reducer class and then Shuffling will take place in which the data which is received from the mapper class will be shuffled and ordered, and then later it will be indexed and stored into the NoSQL databases.



5. Hadoop Runtime Indexing Construction.

While indexing, we used two modes of operations, one with and one without href. We have a map and reducer phase after which we serialize them to JSON and then put them inside HBase. Whichever has the highest count, has the highest score in our Indexing.

6. Hadoop-created index to do the ranking

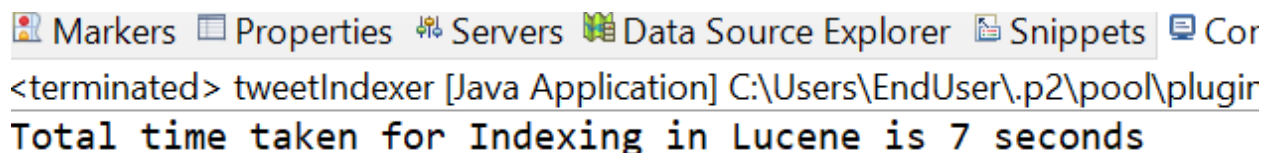
After the indexing is done, we have JSON messages which we pass as a query, and from the searcher function, we get a set of tweets that have the words in them. We then rank them according to the paradigm of TF-IDF and then return the highest scored tweets in descending order.

7. Hadoop Run time vs Lucene Run Time

The run time of Hadoop indexing via constructing lists using MapReduce and then indexing by Tf-IDF is roughly equivalent to the Lucene index creation using the StandardAnalyzer() and IndexWriter() functions. This tallies to somewhere around 9 seconds for a 200 MB file of tweets. This changes somewhat when observing larger file sizes. Using dummy data we simulated a larger file size for MapReduce and noticed a time difference of 10 to 12 percent with Hadoop winning out at larger and larger file sizes and Lucene winning out with relatively smaller file sizes.

The difference between the runtimes is explained by the idea and purpose of map-reduce, which was originally developed to handle large file databases versus Lucene which is not optimized for very large files. This leads to a lower effective time for Lucene when the file sizes are small but causes it to lose out when the data exceeds a certain capacity. The reverse can be seen when it comes to Hadoop, as it is optimized for large files and reports lower runtimes on large datasets.

Indexing in Lucene:



```
Markers Properties Servers Data Source Explorer Snippets Cor
<terminated> tweetIndexer [Java Application] C:\Users\EndUser\.p2\pool\plugir
Total time taken for Indexing in Lucene is 7 seconds
```

Indexing in Hadoop:

```
Total time taken for Indexing in Hadoop is 9 seconds
```

Note: We have tested for smaller size files, so Lucene is faster. If we test for larger data Hadoop is the efficient way.

8. Limitations of the system.

Limitation in Twitter API:

Even though twitter's essential account allows us to pull the tweets for the past seven days it has its own limitation of 900 requests/15 minutes, which gives us a longer time to pull tweets for the required amount of tweets.

Limitation in Lucene:

Lucene cannot handle the large dataset efficiently. We find out this limitation while we are hard testing the data into the Lucene. It has a standard analyzer method that sequentially moves towards each word and tokenizes each and every word so when it comes to large data for indexing Lucene is not a good choice and we can solve this problem by using Hadoop.

Limitation in Apache Hadoop:

Hadoop cannot handle when we give a very small file, It will be very inefficient as the job of map-reduce is to reduce the data into smaller if we give smaller data map-reduce use will not be there

9.Obstacles and solutions:

While crawling there will be an issue of Duplicate Tweets:

Solution: We have found some duplicate tweets have been pulled by the API, in order to rectify the issue we used set() a inbuilt python function to get rid of these duplicates. The very

first tweet will be stored in the tweets and the rest of the tweets will be avoided if the tweets have been already stored in the set.

While crawling we have issues with tweets with both geolocation tagged and untagged tweets:

Solution: Some tweets will have been posted by the user without the geolocation and when we are crawling the tweets it will pull all the tweets with or without the location so we have used the filter criteria to pull the tweets only with location. These tweets with geolocation in one file and tweets without geolocation in another file have been created separately.

While crawling we have issues with tweets that contain emojis and some special character:

Solution: Tweets that contain the emojis and some special characters have been cleared by using our own function cleantext() which get the tweets with emojis and special characters and those will be removed from the text and send back the tweets only with the alphabets and numeric values.

10. Instructions on how to deploy the system.

Deploying Twitter crawler:

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

Running 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.

Running Web application Application:

Requirements: Nodes js and angular, java should be installed.

Angular set up: Install Node js file and give the below command to install angular from below command line.

```
npm install -g @angular/cli
```

In our search engine folder go to the angular -> frontend and go to terminal and give the command Ng-serve then open your web search and give the link localhost:4200 Also make sure to run the spring boot application and don't change the spring boot port from 8080 localhost

Running Hadoop:

Hadoop set up

1. Please need to change the directory where Hadoop is stored

```
$ cd /< directory where Hadoop is stored >/Hadoop-3.3.0/etc/hadoop
```

2. Open Hadoop-env.sh file in order to set up the Hadoop environment.

```
$ vi Hadoop-env.sh
```

copy below line

```
export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
```

3. Open Hadoop-site.xml and add a property to it using below commands

```
$ vi hdfs-site.xml
```

add below line inside configuration tag

```
<property>
<name>dfs.replication</name>
<value>1</value>
</property>
```

4. Open core-site.xml and add a property to it using below commands

```
$vi core-site.xml
```

add below line inside configuration tag

```
<property>
<name>fs.defaultFS</name>
<value>hdfs://localhost:9000</value>
</property>
```

5. Open mapred-site.xml and add below properties to it using below commands

```
$vi mapred-site.xml
```

add below line inside configuration tag

```
<property>
<name>mapreduce.framework.name</name>
```

```

<value>yarn</value>
</property>
<property>
<name>mapreduce.application.classpath</name>
<value>$HADOOP_MAPRED_HOME/share/hadoop/mapreduce/*:
$HADOOP_MAPRED_HOME/share/hadoop/mapreduce/lib/*</value>
</property>

```

6. Open yarn-site.xml and add below properties to it using below commands

```
$vi yarn-site.xml
```

add below line inside configuration tag

```

<property>
<name>yarn.nodemanager.aux-services</name>
<value>mapreduce_shuffle</value>
</property>
<property>
<name>yarn.nodemanager.env-whitelist</name>
<value>JAVA_HOME,HADOOP_COMMON_HOME,HADOOP_HDFS_HOME,HADOOP_C
ONF_DIR,CLASSPATH_PREPEND_DISTCACHE,HADOOP_YARN_HOME,HADOOP_MA
PRED_HOME</value>
</property>

```

6. Hbase setup:-

Change to the conf directory in hbase in order to setup hbase

```
$ cd < directory where hbase is stored >/hbase-2.4.1/conf
```

```
$vi hbase-env.sh
```

Add the below line

```
export JAVA_HOME = /usr/lib/jvm/java-8-openjdk-amd64
```

```
$vi hbase-site.xml
```

add below line inside configuration tag

```

<property>
<name>hbase.cluster.distributed</name>
<value>>false</value>
</property>
<property>
<name>hbase.tmp.dir</name>
<value>./tmp</value>
</property>
<property>
<name>hbase.unsafe.stream.capability.enforce</name>
<value>>false</value>

```

```

</property>
<property>
<name>hbase.zookeeper.property.clientPort</name>
<value>2181</value>
</property>
<property>
<name>hbase.rootdir</name>
<value>file:///home/hadoopusr/HBASE/hbase</value>
</property>
<property>
<name>hbase.zookeeper.property.dataDir</name>
<value>/home/hadoopusr/HBASE/zookeeper</value>
</property>

```

After this we need to change bashrc file

```
$ vi ~/.bashrc
```

copy below lines

```

export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
export HADOOP_HOME=/usr/local/bigdata/hadoop-3.3.0
export PATH=$PATH:$HADOOP_HOME/bin
export HBASE_HOME=/usr/local/bigdata/hbase-2.4.1
export PATH=$PATH:$HBASE_HOME/bin
$ source .bashrc

```

SSH:-

```
$ssh localhost
```

If can't connect to ssh then execute below commands

```

$ ssh-keygen -t rsa -P "" -f ~/.ssh/id_rsa
$ cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
$ chmod 0600 ~/.ssh/authorized_keys

```

Preprocessing:

First we need to start the hadoop daemons. Do this by running start.sh in your hadoop file directory,

Next start hbase by navigating to its directory and then running start-hbase.sh.

Change to the directory where the indexer is stored and build the jar file using mvn clean compile package assembly:single

Then run the jar and invoke the hadoop dependencies as well using the command:

```
$ hadoop-jar /project directory/hbase/Hbaseindexer-jar indexer.jar /< crawled data directory>
```