Build your Search Engine
Group - 9
CS 242 (Winter 2022)

Samin Arefin (862315840)
Md Rayhanul Masud (862317259)
Md Abdullah Al Mamun (862259126)
Akhilesh Reddy Gali (862258042)
Sai Teja Pasupulety (862313257)

November 3, 2022

1 Hadoop indexing Architecture

Hadoop indexing of the search engine comprises a number of interplay between several components of the system. The crawled tweets are processed and indexed using Hadoop. The generated indexing is further used by the web component of the system to serve the ranked results as response to the user defined query.

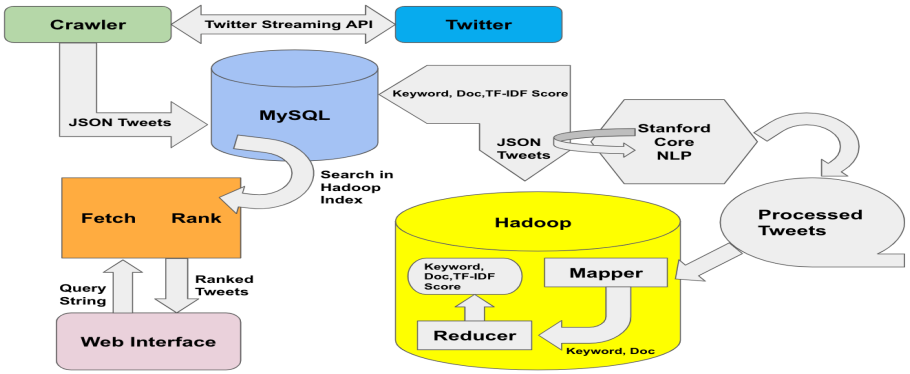

Figure 1: Architecture of Hadoop indexing

A multi-threaded crawler is used to crawl tweets related to a range of selected trending topics using Twitter Streaming API. The crawled tweets are saved in a MySQL database. Since the tweets may contain unnecessary keywords, they are preprocessed using Standford Core NLP Library. The library

removes the stop words, and performs stemming and lemmatization. Then, Hadoop uses a mapper to emit $< keywordfromthetweet, tweet\_id >$. Next, $< keywordfromthetweet, tweet\_id\_list >$ is fed as the input to a reducer. The reducer then calculates the TF-IDF score of the given keyword per tweet where the keyword is found. The generated inverted indexing is stored back into the MySQL database. Whenever, the user searches for tweets providing any query string, the query string is processed and necessary keywords from the query are extracted. They are searched in the database, and a list of documents is returned with their corresponding scores. Finally, the documents are displayed in the web interface according to the computed ranking scores.

2 Details of how hadoop was used

We used hadoop to build the inverted index database of our search engine. The distributed nature of hadoop enabled us to create our inverted index database in a fault tolerant and speedy manner. The mapreduce notion of hadoop is inherently scalable where we could easily change the number of reducer tasks to our preference. Since, the tweets we collected were stored in a MySQL database, it was convenient for us if we also could store the inverted index in the same database.

tweet_id	json
1500564813751201799	{"created_at":"Sun Mar 06 20:11:45 +0000 2022","id":1500564813751201799…
1500564814019448833	{"created_at":"Sun Mar 06 20:11:45 +0000 2022","id":1500564814019448833…
1500564814183030784	{"created_at":"Sun Mar 06 20:11:45 +0000 2022","id":1500564814183030784…
1500564814321664001	{"created_at":"Sun Mar 06 20:11:45 +0000 2022","id":1500564814321664001…
1500564814388625408	{"created_at":"Sun Mar 06 20:11:45 +0000 2022","id":1500564814388625408…
1500564814397317121	{"created_at":"Sun Mar 06 20:11:45 +0000 2022","id":1500564814397317121…
1500564814405509127	{"created_at":"Sun Mar 06 20:11:45 +0000 2022","id":1500564814405509127…
1500564814698926083	{"created_at":"Sun Mar 06 20:11:45 +0000 2022","id":1500564814698926083…
1500564814745464832	{"created_at":"Sun Mar 06 20:11:45 +0000 2022","id":1500564814745464832…
1500564814753632261	{"created_at":"Sun Mar 06 20:11:45 +0000 2022","id":1500564814753632261…

Figure 2: view of the tweet table

To do this, we needed to implement some custom key-value classes that could read from MySQL and write to MySQL. Since, there is no default implementation of database oriented key-values types in the standard hadoop library, we created our own classes that implement hadoop's Writable and DBWritable interfaces. Next, we created our mapper and reducer for the task. The basic structures of Map Reducer is given below in figure 3.

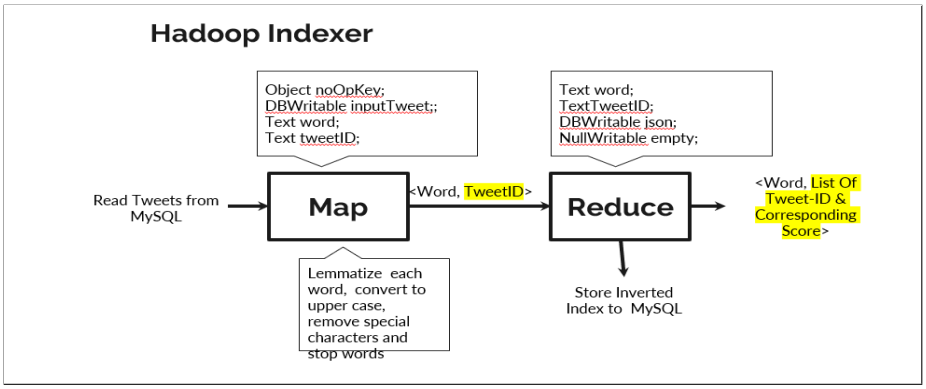

Figure 3: Working mechanism of Map reducer

3 How We use Map reducer in Hadoop

We created one mapper and reducer for our task. The mapper has a signature of $< Object, DBInputWritable, Text, Text >$, where the input value type is DBInputWritable that we have custom implemented to read from MySQL "tweet" table. And, the output key/values types are Text. The mapper reads every row from the "tweet" table which is a table of two columns - tweet_id, and the json value. Then, it extracts the json value using Java Jackson library to get the tweet body. Then we perform an extensive set of preprocessing on the tweet content which is described in detail in the following sections. For every token after preprocessing, we emit the token and the tweet_id. After the mapping job is finished we need to reduce the aggregated values per key from the mapping job output. Our reducer has a signature of $< Text, Text, DBOutputWritable, NullWritable >$ where the incoming key is a single word and the value is a list of tweet_ids emitted from mapper upstream and aggregated by individual word by the reducer downstream. Next, from the list of tweet_ids we computed the TF-IDF score for each of them. Finally, we create a JSON structure where each key represents a tweet_id and the value is TF-IDF score. We write this JSON along with the corresponding word as a single inverted-index row in a table called "tweet_inverted_index" in the MySQL DB.

4 Explanation and justification of the indexes built by hadoop

We ensure indexing by hadoop through Stemming, Lemmatization and Removing Stop Words and also by handling hashtags. Now we will discuss three different features shown in figure 5 that we adopt in our project

word	json_index
@JAVADK64359167	{"1500569316806307844":4.698978690138799}
@SJAVA_ATM	{"1500572389956067336":4.698978690138799}
JAVA	{"1500572373652803584":2.914169685653851,"1500566552558538757":2.914169685653851…
#JAVA	{"1500568311670661121":2.9286301580643452,"1500570482948677633":2.92863015806434…
#JAVASCRIP	{"1500570244607545344":4.698978690138799}
@GUIDESJAVA	{"1500569127517306885":4.221874806503572,"1500570736213344261":4.221874806503572…
@JAVASCRIPTFEED	{"1500566486221754368":3.74480566086871,"1500566326091587585":3.74480566086871,"…
#JAVASCRI	{"1500572373652803584":4.698978690138799}
#JAVASCRIPT30	{"1500569881913274373":4.397957380103888,"1500570047751856130":4.397957380103888}
#JAVASCR	{"1500565162759114752":4.698978690138799}
JAVASCRIPT	{"1500572496600473603":3.119516346164721,"1500570937640558596":3.119516346164721…

Figure 4: view of the tweet_inverted_index table

Figure 5: Stemming, Lemmatization and Removing Stop Words

4.1 Preprocessing: Stemming, Lemmatization and Removing Stop Words

To reduce influential forms and to transform words to their common base form, we used both stemming and lemmatization into our architecture. Often there are many representation of a single word e.g with different kinds of punctuation and simple grammatical variation. Treating them as separate words is often unnecessary and also takes up more space in the inverted index database. To tackle this issue we transformed the tweet body using Stanford CoreNLP library to perform lemmatization and stemming. For each tweet in the database, we extract the JSON text body using Java Jackson library, then we pass it to the Stanford CoreNLP library to do the lemmatization and then stemming. After that, we remove different kinds of stopwords from the string. Removing stopwords is also very important for any information retrieval system as scoring unnecessary words will only slow down the map reduce job. We downloaded a widely used stopwords.txt file from internet (also attached with our code), and used it to remove any words from the tweet text body.

4.2 Handling Hashtags

Stemming and lemmatization work great for various kinds of word in the tweet body. But, hashtags are special in a way that if preprocessed will lose it's actual meaning. That's why, when preprocessing texts, we first tokenized the whole tweet body using PTBTokenizer from Stanford CoreNLP library which is a fast, rule-based tokenizer implementation that produces Penn Treebank style tokenization of English text. After that, we split the tokens into two categories, one that starts with "#" and and others that don't. We only push the non-hashtag tokens into the preprocessing pipeline to do the lemmatization, stemming and stopwords removing. After that, we add the untouched hashtags to the list of pre-processed tokens to get the final list of words.

4.3 Ranking Based on the Hadoop-Created Index

The inverted indexing for the tweets was generated using Hadoop, and for each of the keyword, the TF-IDF score was computed for all of the tweets where the keyword was found. When a user searches for some random string, the query string is pre-processed in the same way used in case of the hadoop index creation. The pre-processed query string provides a list of keyword which are then searched in the hadoop indexing saved in a database.
The scoring strategy is to iterate over each of the keyword to extract the TF-IDF score for each of the tweets related to it, and the score for each tweet (document) is calculated. The sum of the scores by the keywords provides the score for a tweet. Finally, sorting the scores provides the ranking of the documents.

4.4 How Lucene Indexing works with QueryParser to return the result

To detail about the working mechanism of Lucene indexing that works with QueryParser to return the result can be divided into following several parts. Now we provide overview of those parts below.

4.4.1 Lucene Indexing

The implementation of the project experimented over various combination of different useful features to better understand the trade-off between indexing any number of features and indexing the most useful ones. The final implementation considered tweet text and hashtags to be the most significant features for ensuring good user experience.

4.4.2 Pre-processing Parsing Query

The query string given by the user may contain irrelevant keywords (e.g. this, that, the) and stop words. On the contrary, during the indexing creation by Lucene, the indexing elements (e.g. tweet text) were pre-processed using the Standard Analyzer API available in Lucene, though hashtags were not treated

the same way due to its different domain specific importance. As a result, the removed keywords will not be found in Lucene indexing. To ignore unnecessary keyword searching, the query string was also pre-processed using the same analyzer.

5 Ranking the Search Results using TF-IDF score While Using Lucene

Lucene comes with a variant of TF-IDF (Term Frequency-Inverse Document Frequency) as its default scoring mechanism. Though it supports many of the other widely used scoring techniques, our implementation opted for the default one. As mentioned earlier, the indexing was performed using tweet text and hashtags, the keywords parsed from the query string were searched in the generated Lucene indexing. The best ranked results were returned by the available API. Usually, Lucene query searching offers the basic query facilities known as **Basic Query**. On top of that, Lucene supports a number of diverse query capabilities. In this project, in addition to **Basic Query**, two more querying capabilities were introduced in the implemented search engine. They are **Boolean Query** and **Term Boosting Query**. To prevent unrelated bulk of search results, filtering criterion was adopted.

5.1 Run time of the Hadoop index construction Process

For the hadoop job we experimented with different number of tweet dataset size to see how the runtime varies between different size of dataset. We took batches of tweets in incremental sizes of 10k, 20k, 30k, 40k and 50k and ran the hadoop job 5 times to compare the time. Figure 3 shows the variation of between different size of tweet dataset.

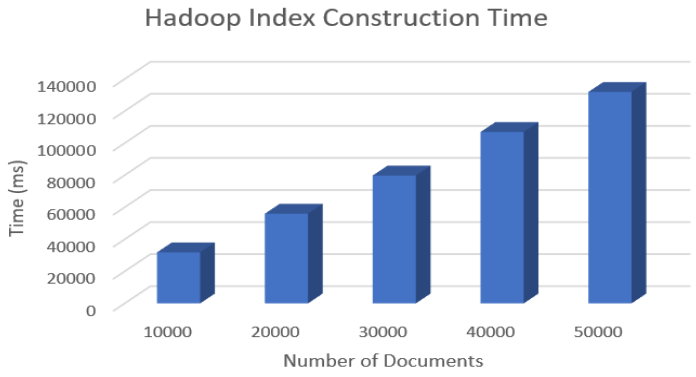

Figure 6: Running time of hadoop index creation

5.2 Run time of the Lucene index construction process

The project implementation experiments over indexing of different fields from twitter dataset. The more fields are indexed, the more index creation time is required. On the contrary, it is obvious that multiple field indexing provides greater flexibility. In addition, searching over multiple indices incurs more response time than searching less indices. So, the experimentation derives a conclusion regarding indexing and searching that there should be a trade-off maintained for choosing fields for index creation and searching. Figure 7 shows the index creation time by Lucene using tweet text and hashtag. The figure suggests that the more documents are indexed, the more running time is consumed.

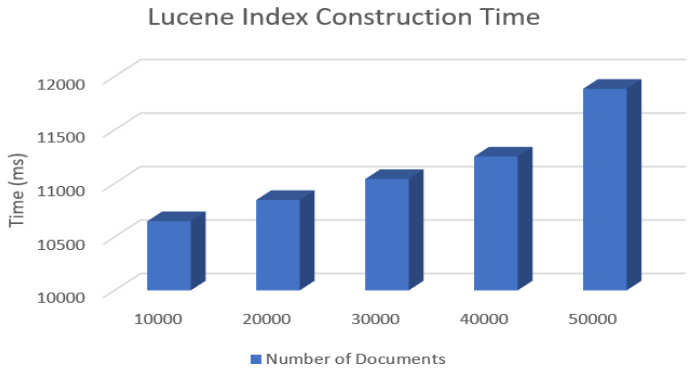

Figure 7: Running time of indexing with tweet text and hashtag

5.3 Comparison between the run time of the Lucene index construction from report A and hadoop Indexing

We have run two index creation pipeline one with hadoop and another with Lucene. Both were done using same tweet and hashtags and on same number of documents(10k-50k). From the runtime charts of both the system, it is obvious that for same number of documents, the hadoop process is taking a much longer period of time than the Lucene version. There can be multiple reasons for that, hadoop is optimized for a clustered setup of networked servers for generic computation where Lucene is solely made for the task of full text searching and analysis. Lucene takes better advantage of the local processor cores and performs many optimization for text analysis. Since, we used hadoop on a single node setup, the distributed advantage of hadoop was not reflected on our benchmarks and indexing.

6 Limitations of system.

- *Limitation 1:* The response of the implemented search engine is quite slower compared to the search engine we use in our everyday life. In spite of performing optimization during calculation of results, there was room for improvement in the overall effectiveness. Consequently, ranking could be improved.

- *Limitation 2:* Hadoop was found to be slower compared to Lucene. It is because Lucene is armed with in-built super optimized and fast indexing mechanism, whereas the search engine built powered by Hadoop itself followed our defined optimizations and ranking procedures.

- *Limitation 3:* As our system indexes based on the keyword, so it will return the result based on the presence of the keyword only. It does not provide any semantically similar result. For example, this system treats the "football" and "soccer" keyword different but they are semantically same.

- *Limitation 4:* As our crawling system crawls the data and indexes the data based on the data. However, the original tweet may change and in that case if the newly updated version of tweet is stored, it does not replace the previous tweet. But both tweets can be returned by our search engine if the query keywords are available in both tweets. Besides, it may slow down the performance of the search engine incurring more response time.

- *Limitation 5:* Lucene indexing can utilize at most one index writes at any instant. This little limitation degraded the performance to some extents.

7 Obstacles and solutions

While implementing the projects we came across some obstacles. Now we discuss the details about them.

- **Obstacle 1:** The twitter streaming API allows only one live connection per IP address. As a result, it is tough to consume more data in parallel. However, we have tried with multiple consumer processes to consume more data. But almost all our experiments were severely rate limited by twitter except using two simultaneous connections at a time. But the fact is that the two consumers sometimes consume the same tweet and that is why we used bloom filter to avoid the duplicate tweet consumption.

- **Obstacle 2:** General search API allows 500K tweets crawling per month. If the amount was more relaxed, we could have recursively crawl the data feed using more parallelism than the current solution.

8 Screenshots showing the system in action

Now we provide the implementation details of our code for index Creation using hadoop and Lucene. We show the code snippet of our Hadoop Tweet Mapper implementation in figure 8.

```
public static class TweetMapper extends Mapper<Object, DBInputWritable, Text, Text> {
    private final Text word = new Text();
    private final Text id = new Text();

    public void map(Object key, DBInputWritable value, Context context) throws IOException, InterruptedException {
        JsonNode jsonNode = OBJECT_MAPPER.readTree(value.getJson());

        String text = jsonNode.path("text").asText();
        String docId = jsonNode.path("id_str").asText();

        id.set(docId);

        for (String token : processTextToTokens(text)) {
            word.set(token);
            context.write(word, id);
        }
    }
}
```

Figure 8: Implementation of Tweet Mapper in hadoop

Next, we show the code snippet of our Hadoop Tweet Reducer implementation in figure 9.

```
public static class TweetReducer extends Reducer<Text, Text, DBOutputWritable, NullWritable> {

    public void reduce(Text key, Iterable<Text> values, Context context) throws IOException, InterruptedException {
        Map<String, Integer> docIdMap = new HashMap<>();
        Map<String, Double> scoreMap = new HashMap<>();

        for (Text value : values) {
            docIdMap.compute(value.toString(), (__, oldVal) -> oldVal == null ? 1 : oldVal + 1);
        }

        double idf = Math.log10(1 + DATA_SIZE / docIdMap.size());

        docIdMap.forEach((k, v) -> scoreMap.put(k, v * idf));

        context.write(new DBOutputWritable(key.toString(), OBJECT_MAPPER.writeValueAsString(scoreMap)), NullWritable.get());
    }
}
```

Figure 9: Implementation of Tweet Reducer in hadoop

Now, we show the code snippet of our Lucene Index creation implementation in figure 10.

Figure 11 shows the code snippet of Web Controller.

Figure 16 shows graphical user interface (GUI) while searching keyword using Hadoop Indexing.

Figure 13 shows graphical user interface (GUI) while searching keyword using Lucene Indexing.

```
public void createIndex(List<TweetJson> tweets) throws IOException {
    // Iterate over each tweet to create indexing
    for (TweetJson tweet : tweets) {
        Document doc = new Document();

        if (tweet.getTweetText() != null) doc.add(new TextField(TEXT, tweet.getTweetText(), Field.Store.YES));
        if (tweet.getUsername() != null) doc.add(new StringField(USERNAME, tweet.getUsername(), Field.Store.YES));
        if (tweet.getTimestamp() != null) doc.add(new TextField(TIMESTAMP, tweet.getTimestamp(), Field.Store.YES));
        if (tweet.getUserLocation() != null)
            doc.add(new StringField(USERLOCATION, tweet.getUserLocation(), Field.Store.YES));

        if (tweet.getHashTags().size() > 0) {
            String hashTagString = String.join( delimiter: ", ", tweet.getHashTags());
            Field hashtag = new StringField(HASHTAG, hashTagString, Field.Store.YES);
            doc.add(hashtag);
        }

        if (tweet.getCoordinates() != null)
            doc.add(new StringField(COORDINATES,  value: tweet.getCoordinates()[0]
                    + ", " + tweet.getCoordinates()[1], Field.Store.YES));

        if (!tweet.getUrls().isEmpty()) {
            String urlString = String.join( delimiter: ", ", tweet.getUrls());
            Field urlField = new StringField(URL, urlString, Field.Store.YES);
            doc.add(urlField);
        }

        this.indexWriter.addDocument(doc);
    }
}
```

Figure 10: Code snippet of Lucene Index creation

```
@RequestMapping(value = "/search", method = RequestMethod.GET)
public @ResponseBody
List<Tweet> search(
        @RequestParam(name = "queryString") String queryString,
        @RequestParam(name = "requestType") String requestType) throws Exception {

    long start = System.currentTimeMillis();
    List<Tweet> tweets = new ArrayList<>();
    if (requestType.equalsIgnoreCase(LUCENE)) {
        ScoreDoc[] rankedDocs = SEARCH_ENGINE.searchIndex(LuceneSearchEngine.SINGLE_FIELD_QUERY, queryString);
        tweets = SEARCH_ENGINE.showResult(rankedDocs);

    } else if (requestType.equalsIgnoreCase(HADOOP)) {
        tweets = tweetService.findAllByWord(queryString);
    }

    long end = System.currentTimeMillis();

    tweets.add(new Tweet(String.valueOf(end - start),  user: ""));

    return tweets;
}
```

Figure 11: Code Snippet of Web Controller

9 Database and Web-Server

We have stored the crawled tweets and inverted index both into **MySQL**. The tweet table contains two columns - the tweet_id and the JSON column containing the whole tweet as a single json object. And the inverted_index table contains a column for word and another for the corresponding inverted index. The latter one is also a JSON object where every key in the JSON is a tweet_id and the value is the **TF-IDF** score. For the web-server implementation we have used spring boot using Java and Gradle as build system. The framework uses an embedded **Tomcat server** to easily deploy the whole web application using a fat-jar file. For templating, we have used JSP templates along with jquery and bootstrap.

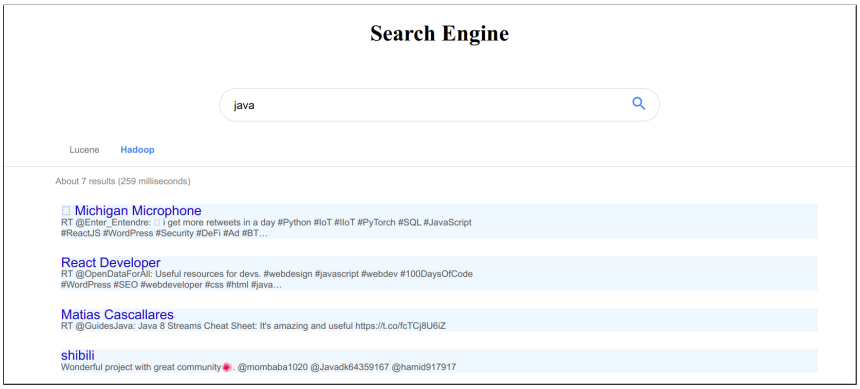

Figure 12: Demonstration of search engine: Indexing by Hadoop

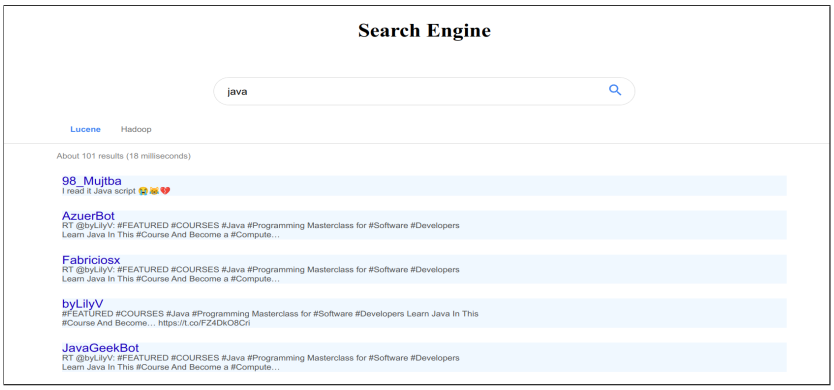

Figure 13: Demonstration of search engine: Indexing by Lucene

10 Higher Level Summary of our Project

We sought to build a search engine based on twitter data indexing by Lucene and hadoop, in which we considered only geotagged tweets while gathering data using Twitter Streaming API. The overview of our project is given below:

Mainly we can divide our project into two parts which are described below elaborately:

- **Crawling Tweets:** For this project, we decided to crawl our data from twitter as it provides us with a large and diverse scope of data collection platform. We used Twitter data to create a search engine using Lucene Indexing that allows users to search for any keywords related to Covid, Cloud, Information, Programming, Java, Amazon, Google, Apple, California, and what's going on in the world right now.
- **Index the data using Lucene and hadoop:** After crawling the

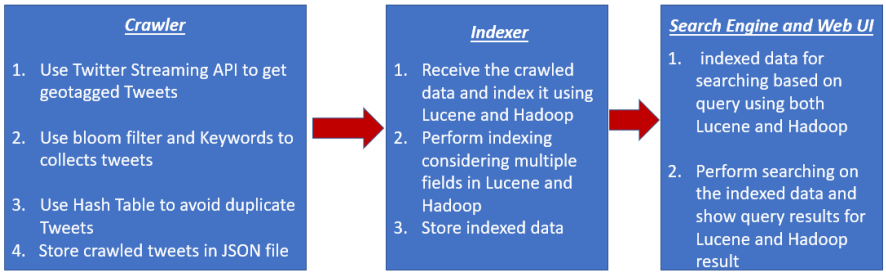

Figure 14: Overview of the project

data from twitter, we need to perform indexing for the search engine. As searching through individual pages for keywords and topics would be a very slow process for search engines to identify relevant information, we use indexing. Many search engines including Google use indexing. One of the most common indexing techniques is to use inverted indexes also known as reverse index. An inverted index is a system wherein a database of text elements is compiled along with references to the documents which include those elements. Then, search engines use a process called tokenization to reduce words to their core meaning, thus reducing the amount of resources needed to store and retrieve data. This is a much faster approach than listing all known documents against all relevant keywords and characters. Lucene and hadoop are used for indexing in this project which provides indexing and search features, as well as spell checking, hit highlighting and advanced analysis/tokenization capabilities.

11 Instructions on how to deploy the system

There are two projects attached with our report. One is "CS242-PartB-Hadoop" and another "CS242-PartB-Web". Hadoop and Web both depend on a running MySQL server. We have locally used MySQL version 8 on a docker container. To start either one of the project there must be a running MySQL DB with a database schema called "CS242" and two tables created beforehand. The DDL schema is shown in the following figures. With that, we can *cd* into the hadoop project and execute ./start_hadoop_job.sh shell script to launch the hadoop job to create the inverted index. After successful finish, the tweet_inverted_index table will be populated with the corresponding result. Finally, we can jump into the web project. First we need to create the lucene indices. We can execute the ./start_lucene_job.sh shell script to populate the lucene indices. After that we need to execute ./start_web_server.sh to start the web server. We are omitting the tweet.txt raw crawled file as it is too large to upload.

```
create table tweet
(
    tweet_id varchar(100) not null primary key,
    json     text         not null,
    constraint tweet_id unique (tweet_id)
);
```

Figure 15: Tweet Table Structure

```
create table tweet_inverted_index
(
    word       mediumtext not null,
    json_index mediumtext not null
);
```

Figure 16: Inverted Index Structure

12 Collaboration Details of Group Members

Member 1: Samin Arefin

- Learned how Hadoop works
- Designing for hadoop indexing strategy
- Apply input data to Map reducer
- Configured Hadoop to work on local system
- Worked on project report

Member 2: Md Rayhanul Masud

- Learned how Hadoop works
- Query Search to retrieve top search results using hadoop indexing
- How to optimize the hadoop indexing code
- Create GUI for performing query
- Worked on project report

Member 3: Md Abdullah Al Mamun

- Analysis and design for hadoop indexing

- Helped in setting up database
- Tweet indexing implementation using Hadoop
- Checked out the optimization procedures for the hadoop indexing code
- Worked on project report

Member 4: Akhilesh Reddy Gali

- Researched the Basic layout of the search engine for Hadoop
- Helped in designing for hadoop indexing strategy
- Implementation of database linking up
- Worked on project report
- Create GUI for performing query

Member 5: Sai Teja Pasupulety

- Visualized the Hadoop in real time
- Learnt how hadoop works
- Helped in setting up database
- How to optimize the hadoop indexing code
- Worked on project report