CS 242: Information Retrieval & Web Search

TV Series Search Engine

PART B(Hadoop Indexing)

Group 20:

Member Information

Aishwarya Pagadala
Avinash Reddy Kummeta
BalaSanjana Thumma
Prudhvi Manukonda
Shivaji Reddy Donthi

COLLABORATION DETAILS

Prudhvi Manukonda: Worked on building inverted indexes using Hadoop.

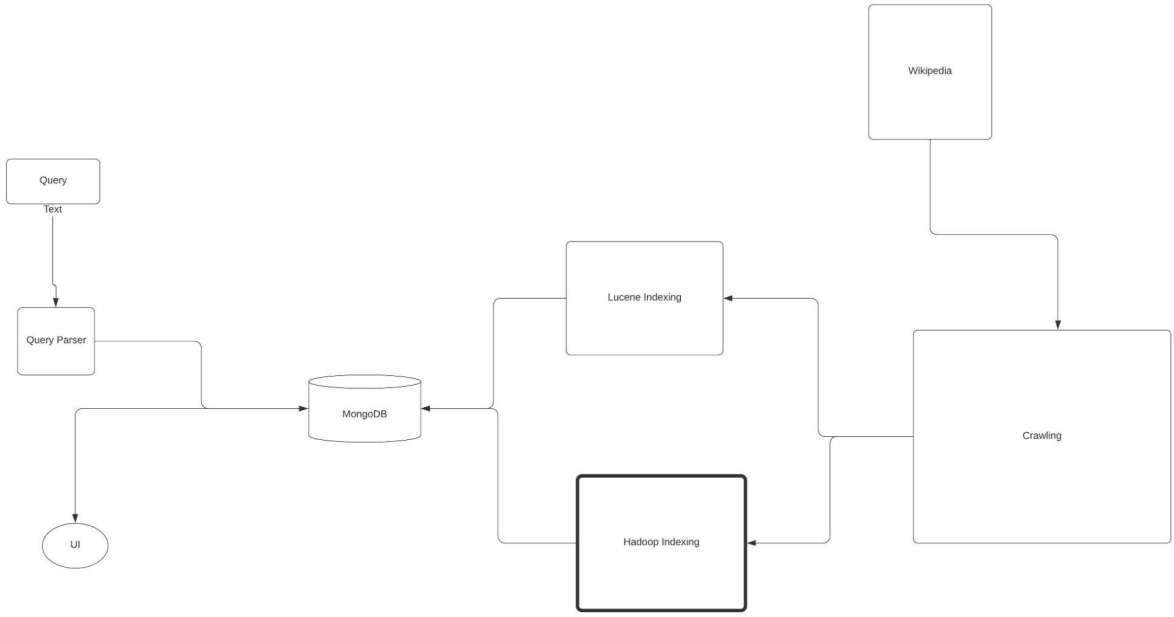
Sanjana Thummala: Worked on building Inverted indexes using Hadoop and worked on pushing data to MongoDB.

Avinash Kummeta: Worked on designing the back-end of the project for parsing the query and ranking the documents using Node.JS.

Shivaji Reddy Donthi: Worked on designing the front-end part using React.JS framework.

Aishwarya Pagadala: Worked on displaying Lucene inverted index results in the back-end using Node.JS and also worked in building the front end.

SYSTEM ARCHITECTURE:



The above figure displays the system architecture, Initially we crawl the Wikipedia page and then we perform map-reduce indexing and Lucene indexing on the crawled output file, we store the inverted index in MongoDB database, and also we store document information in MongoDB. We Built a UI using React.js, in which the user enters the query, In the backend, we search the query in the inverted index stored in the MongoDB and display the corresponding results in UI.

OVERVIEW OF THE MAP-REDUCE:

Map-Reduce has two parts: Mapper Function and Reducer Function. After crawling the data we get output in the form of a text file. We have a set of stop words loaded and loaded in a HashSet. After reading the data from the document we check whether this wor present in HashSet, If Present **we removed those stop words** to increase the efficiency of the search engine and to reduce the load of the database. In addition to it, we have read the document name in which the current word is present. Mapper gets input in the form of <key, value> pairs. In mapper, **we have done stemming** to ameliorate the efficiency of the search engine. From Mapper, we emit **<word,doc_name>** to reducer. So reducer gets the corresponding pair which have the same word.

After mapper emission to the reducer, In reducer, we get input as <term,doc_name> in which all have the same terms. We store document_name and its term frequency in the hashmap. For all keys in hashmap, we append to a new_string which is written below

docValueList.append(doc_name+””+map.get(doc_name)*Math.log(total_Documents/HashMap.size())

Then we emit **<term,docValueList>** which means we are emitting **<term,doc_name,tfidf>** for all the documents in which that term appears. This tf*idf is later used for ranking.

Mapper code Illustrated below

```
  public static class TokenizerMapper
       extends Mapper<Object, Text, Text, Text>{
    private Text word = new Text();

    PorterStemmer porterStemmer  = new PorterStemmer();

    public void map(Object key, Text value, Context context ) throws IOException, InterruptedException, FileNotFoundException {
      /* to load document name */
      String docName = ((org.apache.hadoop.mapreduce.lib.input.FileSplit) context.getInputSplit()).getPath().getName();
      String value_raw = value.toString().replaceAll("\\p{Punct}"," ");
      value_raw = value_raw.replaceAll("\\d"," ");
      /*reading stop words file */
      Scanner file = new Scanner(new File("stop_words_english.txt"));

      Set<String> wordSet = new HashSet<>()
      while (file.hasNext()) {

       wordSet.add(file.next().trim().toLowerCase())
       }



      // Reading input one line at a time and tokenizing by using space, "'", and "-" characters as tokenizers.
      StringTokenizer itr = new StringTokenizer(value_raw, " '-");
      // Iterating through all the words available in that line and forming the key/value pair.
      while (itr.hasMoreTokens()) {
        // Remove special characters
       itr1= itr.tostring();
       if(wordset.containsKey(itr1){
          continue;}

        word.set(itr.nextToken().replaceAll("[^a-zA-Z]", "").toLowerCase());
        /* stemming the token here */
        porterStemmer.add(word.toString().toCharArray() , word.toString().length());
        porterStemmer.stem();
        word.set(porterStemmer.toString());

        //System.out.println(word.toString());
        if(word.toString() != "" && !word.toString().isEmpty()){
          context.write(word, new Text(docName));
        }
      }
  }
}
```

Reducer code Illustrated below

```
public static class IntSumReducer
     extends Reducer<Text,Text,Text,Text> {
  /*
  Reduce method collects the output of the Mapper calculate and aggregate the word's count.
  */
  public void reduce(Text key, Iterable<Text> values,
                     Context context
                     ) throws IOException, InterruptedException {

    HashMap<String,Integer> map = new HashMap<String,Integer>();
    /*
    Iterable through all the values available with a key [word] and add them together and give the
    final result as the key and sum of its values along with the DocID.
    */
    for (Text val : values) {
      if (map.containsKey(val.toString())) {
        map.put(val.toString(), map.get(val.toString()) + 1);
      } else {
        map.put(val.toString(), 1);
      }
    }
    StringBuilder docValueList = new StringBuilder();
    for(String docID : map.keySet()){
      docValueList.append(docID +"    " + (map.get(docID)*Math.log(500/map.size())));
      System.out.println("    ");
    }
   // docValueList.deleteCharAt(docValueList.length());
    context.write(key, new Text(docValueList.toString()));
  }
}
```

Ranking Using Hadoop

In Lucene it automatically ranks the documents, Whereas in Hadoop we have to rank the documents, So we have stored tf*idf in the inverted index. Initially, we did stemming
For the query, we had done **AND** logic for terms in the query & took the documents in which all words in the query occur and took **the average tf_idf score of every term present in the document and ranked according to that average score in the backend Using Node.Js.**

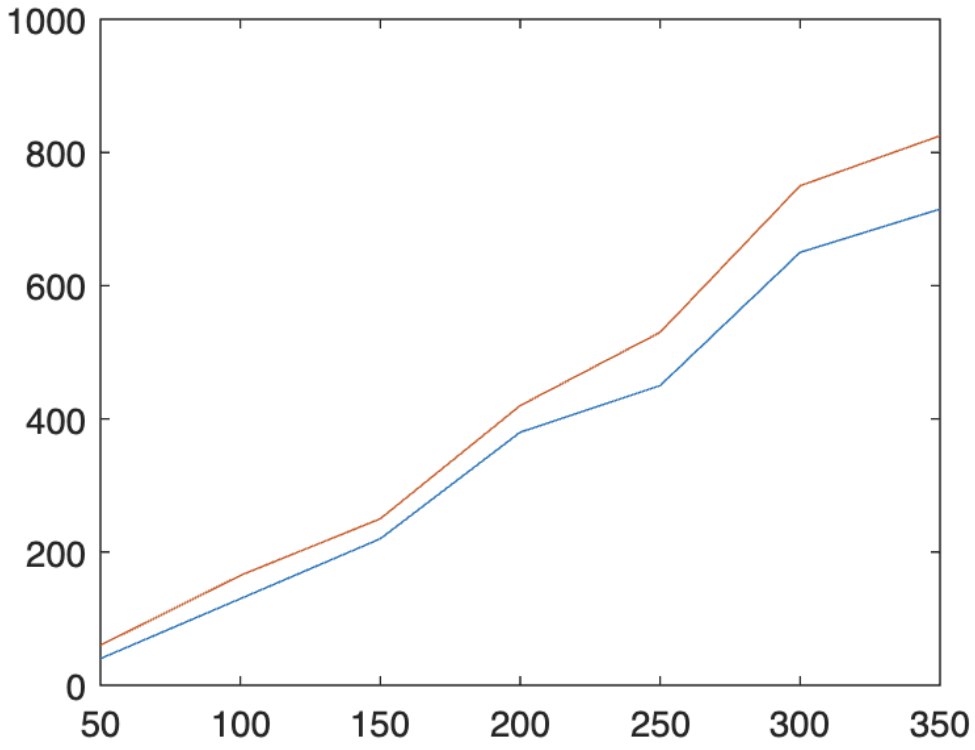
Query results using Lucene:

We used FSDirectory to write the created Lucene indices to an output folder. This folder directory is then used as an input to the DirectoryReader and IndexSearcher, using which we find the document scores for a given query with the help of Query Parser
Here, we take a search query string from the user, using which we find the top 100 documents in their increasing order of score. Since we are indexing a particular document using more than one field, i.e title, paragraph, and file path, we used MultiQueryparser to retrieve the results for the search query with respect to all title and paragraph indexed fields. We did not use the file path as a parameter to the MultiQueryParser search field. We used it to retrieve the original document link path, using which we can go to the document which has the high scores and then return the result in JSON format to the client web interface.

Runtime Analysis or Hadoop Index Construction:

In our project, we have implemented an inverted index both using MapReduce and Lucene. Mapreduce takes more time compared to Lucene since it has to rank the documents and has to perform stemming. Search results of map-reduce and Lucene are demonstrated in a tabular form and also plotted a graph denoting both Hadoop and map-reduce timings vs the number of documents.

Total No: of Documents	Lucene Indexing	Hadoop Indexing
50	40	60
100	130	165
150	220	250
200	380	420
250	450	530
300	650	750
350	715	825



On Xaxis: Number Of Documents, On Y-axis: Time, Red: Hadoop, Blue: Lucene

LIMITATIONS OF SYSTEM:

- We are unable to correct the queries if there is any typing mistake in the query, System only search for the words present in the query.
- We used only used tf*idf frequency for ranking it doesn't work well in all cases. In some cases, although it has more tf_idf it may be less relevant to the user.
- When the user searches for multiple words without a delimiter or space separation between them it will search as a combined word. For example, if we search for HelloWorld search engine takes HelloWorld as a single word and searches since there is no HelloWorld it will not display any results.
- The system doesn't take into account terms dependencies and searches for every term in the query which makes unnecessary computations.
- The system doesn't take into account if one term has higher priority than another term in queries instead it averages the score and gives equal importance to words.

OBSTACLES:

There were some obstacles we faced while doing the project, for some of the problems we are able to come up with the solution.

- Initially, we tried to install Hadoop in the local machine for setting the environment variables, and setting the Hadoop environment took us so much time, So later we realized to use the assigned machines.
- We faced some problems in debugging the MapReduce code because every time for every error we have to edit the code and the program should run on the whole input it took around 5-10 minutes for code to run, So to reduce the time problem we have took small input file and ran the program until we get zero errors.
- We encountered some exceptions like a malformed exception and socket timeout exception which will stop the application so we did some checks to the URL using URL package which checks whether the URL is valid and can be parsed.

Instructions to Deploy

Deploying Hadoop and Run Map-Reduce

- Unzip Hadoop_Index.zip.
- In Hadoop_Index\input files, you will find input files.
- Push all these files to Hadoop setup.
- Give this input file to input in hdfs using the following command

    - hdfs dfs -put /LOCALPATH/input.txt /NEWFOLDER/input/
- Run all .java files using the following command.
    - javac *.java -cp $ (Hadoop classpath)
    - jar cf ii.jar *.class
    - hadoop jar wc.jar WordCount /NEWFOLDER/input /NEWFOLDER/output
    - hdfs dfs –tail /NEWFOLDER/output/part-r-00000

Running frontend part of the project.

Front end is built using React JS So to run start the frontend from the zip

- Go to the folder ReactUI /Search-Engine
- Then Run npm install to install the react package and then npm start.
- Then you can access the frontend part -> http://localhost:3000/

Running the backend part of the project

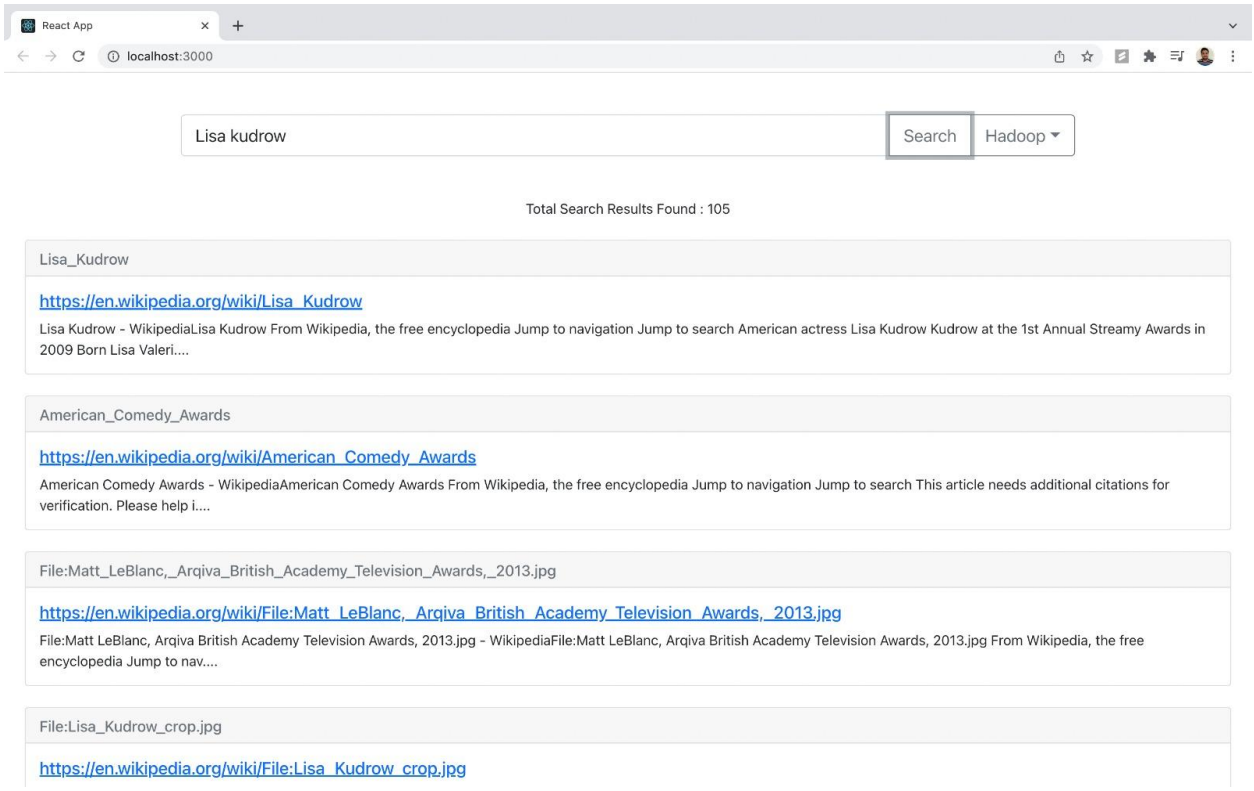
Backend is built using Node Js so to start the backend server from the zip

- Go to the folder serverIR/backend.
- Then run ‘npm install’ to install the node packages
- Then run ‘npm server’ which will start the server at http://localhost:4001

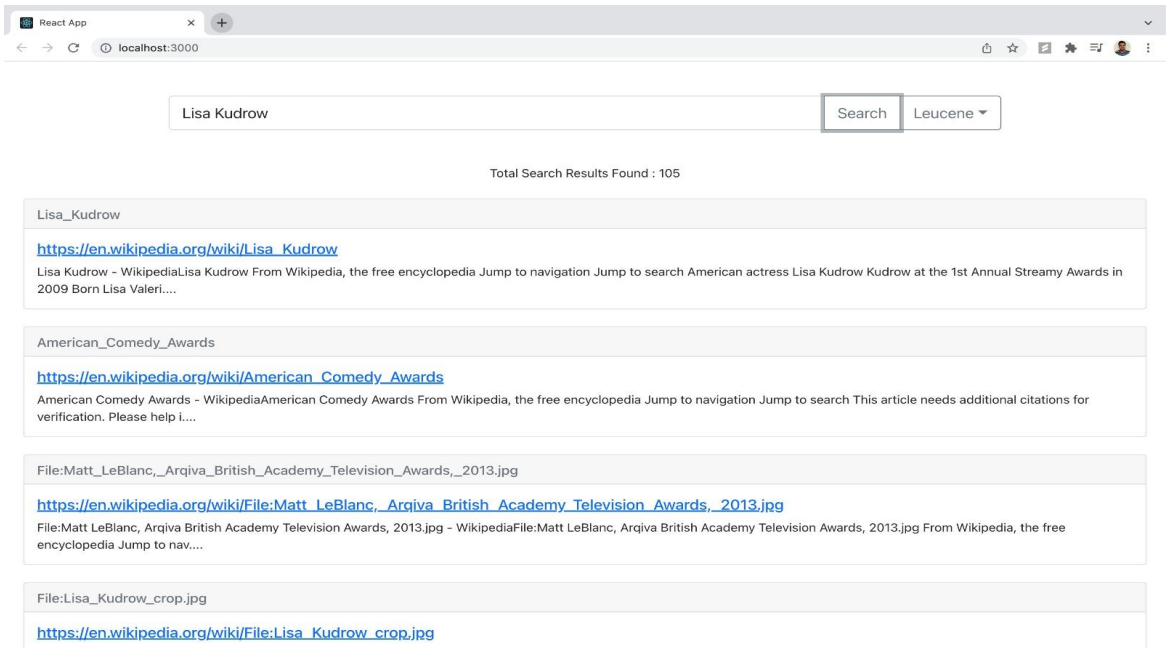
System Demonstration:

1.Home Page

2.Hadoop results



3. Lucene Results



References:

- Tutorial: Intro to React – React (reactjs.org)
- https://medium.com/analytics-vidhya/word-count-using-mapreduce-on-hadoop-6eaefe127502#.
- Welcome to Lucene Tutorial.com - Lucene Tutorial.com
- Node.js Tutorial (w3schools.com)
- MongoDB Tutorial (tutorialspoint.com)