**INFORMATION RETRIEVAL AND WEB SEARCH PHASE 2**

Submitted by,

MOHIT MORARKA (861130666)

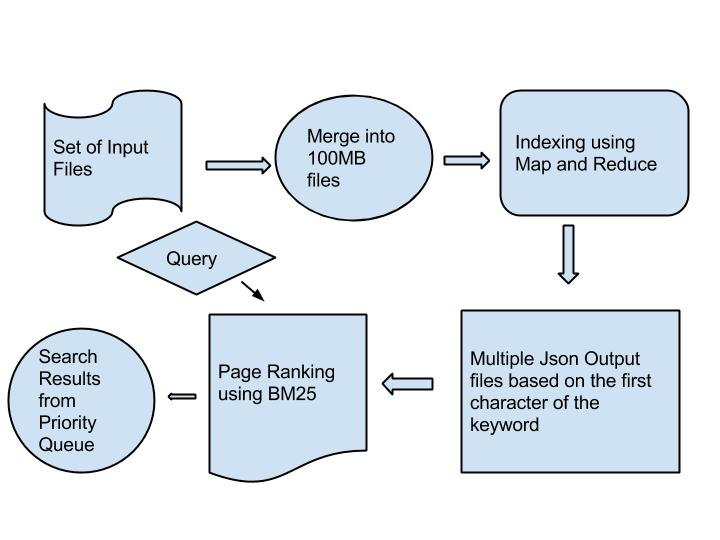
PRASHANTH PONNADI (861135032)

**1> Collaboration Details**

|  |  |
| --- | --- |
| **MOHIT** | **PRASHANTH** |
| Indexing of Hadoop | Searching on Lucene Indexes |
| Splitting the Hadoop output to multiple files using Hash Table | Doing Page Ranking on Lucene Indexes |
| Covert the Hadoop data into JSON format | Support Boolean and multiword queries |
| BM25 Page Ranking for getting top ranked results | Parsing the Hadoop Json Indexes for query retrieval |
| Web Interface using PHP and Wamp Server | Stemming and Stop word implementation on Lucene |
| Stop word implementation in Hadoop indexing using Hash set of stop words. |  |

**2> Overview of the System**

**Architecture-**



**Hadoop Indexing Strategy**

1) **Merge the crawled input files**-

We take the crawled input files generated by our lucene in the first part and merge those files into files of size 100 MB using the Hadoop hdfs getmerge functionality.

The reason we merge these files is Hadoop works better on files of larger size rather than many small files.

**2) Using Map Reduce-**

We input the merged crawled files into the Mapper

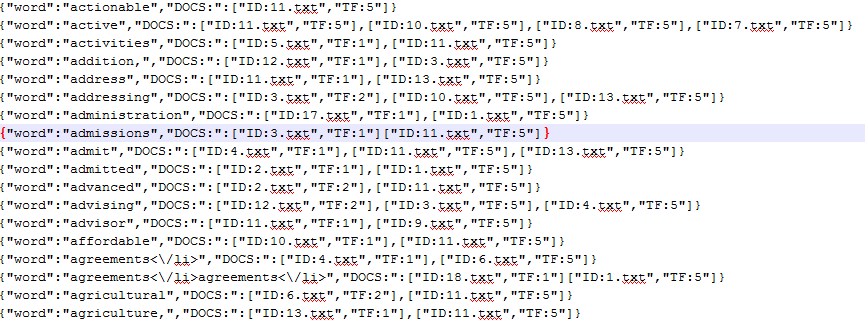
**Output of Mapper**- {word, document ID, Term Frequency}

Here the **key** is the **word** and **value** is the **(document ID, Term Frequency)**

**Output of Reducer**- {word, (docId’s: tf)}

I have kept the code for the Map Reduce at the end for your reference.

**A snapshot of results generated by Hadoop Map Reduce**

****

**3) Split the Hadoop generated index-**

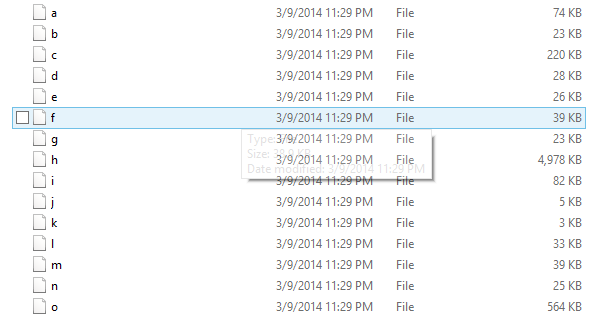
We store the Output of the reducer which is there in JSON format using the technique of splitting the input files based on the first character. So we get the output into 26 files starting from A-Z.

The advantage with this approach is during the searching part we can check for the first character of the query and look into one indexed file rather than looking into all the indexed files.

The problem with this approach is once the data size starts increasing these 27 indexed files become very large so the approach we took is we make use of the hash table to store the indexes in the key value format, with key as the term and value as (document ID, Term Frequency).

We did not store this hash table in the memory because in that case our indexer and searcher would not work as separate pieces. So we store this Hash Table on the disk.

**Snapshot for our split index results-**



**4) Page Ranking using BM25**

We implemented page ranking using BM25 model.

partialRank = Math.log10(n/ni - 1 ) \* (k1 + 1) \* f1 / (k +f1) \* (k2+1) \* qf1 / (k2 + qf1);

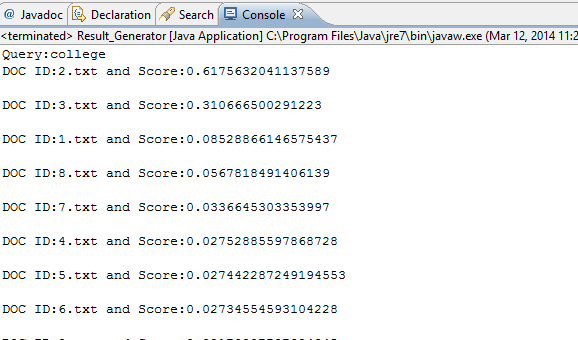
For a given query we first check which indexed files in Json format and parse only one file with filename as the first character of the search query.

We take out the lists of docs and the term frequency for that term by parsing the Json File.

Then we add the BM25 page rank to each of those pages which we found during indexing.

Finally we store the pages along with their page rank in a priority Queue and use its poll method to display the results with higher page ranked results on the top.

**Snapshot for our Page Rank results-**

****

**5) Handling stop words for Hadoop indexing-**

For this we created a Hash Set of GoogleStop words. We create a pattern object for regular expressions, Then we pass it in our matcher which removes the term if it belongs hash set of stop words we had created, removes the term if it starts with non-letters, digits, space or any kind of special characters.

**6) Stemming for Hadoop indexing-**

We made use of the porter stemmer to handle the terms which belong to the same stem.

Porter stemmer places all the terms in the same stem close to each other in indexing because if a particular term has been searched there is a high probability that the next word to be searched belongs to the same stem.

**3>Lucene Index to return the results**

**1) Options Used with Query Parser-**

We check for four fields to be passed in Query parser-

1. Content
2. Title
3. URL
4. Hash Tags

We assign a boost factor to these four fields.

**2) Ranking and returning the results-**

We make use of TopScoreDocCollector API where we specify the number of results to return in our query results.

We then calculate the number of hits based on the score of the top documents and return the top ranked results.

**3) Multiple fields with Query Parser –**

We made use of MultiFieldQueryParser , which behaves similar to the QueryParser but also allows search over multiple fields.

We provided the support for the fields Content, Title, URL and hash tag with our Query Parser.

**3) Multiple word query with Lucene –**

We make use of Boolean query for handling multiple words .

We first put each object into the Term Object which in turn is put in to the prefix query and then all the prefix queries are added to the Boolean Query

BooleanQuery booleanQuery = new BooleanQuery ();

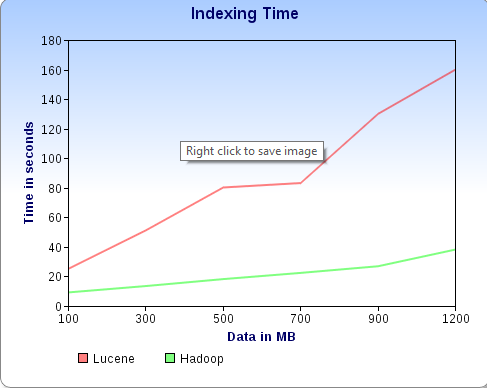
For (String token: tokens) {

booleanQuery.add (new PrefixQuery (new Term (fieldName, token)), Occur. MUST);

}

**4> Run time of Hadoop Index Creation**

|  |  |  |
| --- | --- | --- |
| **Data Size in MB** | **Lucene time in secs** | **Hadoop time in secs** |
| 100 | 25.23 | 9.21 |
| 300 | 51.28 | 13.58 |
| 500 | 80.20 | 18.20 |
| 700 | 83.28 | 22.40 |
| 900 | 130.7 | 26.90 |
| 1200 | 160.9 | 38.49 |



**Discussion-**

We have calculated this graph with crawled data of size 100 MB to 1200 MB.

As we can see from the figure Index creation time for Hadoop is much lesser as compared to Lucene Indexing time.

This can be understood from the reason that in Hadoop all the jobs are happening in parallel and it is there on a distributed node cluster.

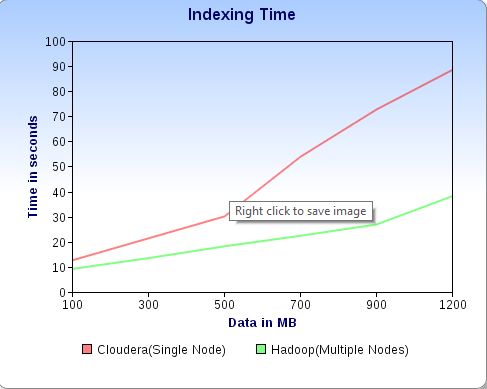
Map Reduce allows for distributed processing map and reduction operations

Map reduce processes the data in a parallel fashion across huge data sets using a large number of nodes

Map reduce takes the advantage of processing large data sets with a parallel distributed algorithm on a cluster whereas in case of Lucene indexing it is sequential.

**5> Comparison of Hadoop (multiple nodes) with single node indexing**

|  |  |  |
| --- | --- | --- |
| Data Size in MB | Cloudera time in secs | Hadoop time in secs |
| 100 | 12.65 | 9.21 |
| 300 | 21.38 | 13.58 |
| 500 | 30.10 | 18.20 |
| 700 | 53.88 | 22.40 |
| 900 | 72.70 | 26.90 |
| 1200 | 88.5 | 38.49 |

****

Discussion-

I did all my development on the cloudera VM with a single node cluster so I was curious to know if we compare its indexing time with a muti-node cluster.

When we started with smaller data size we had very similar run time for both the system but when we started increasing the data size run time of Hadoop on multi node cluster was much smaller as compared to the one on single node, This can be explained from the reason that Hadoop makes use of distributed environments with many nodes in order to enhance its parallel computation.

**6> Web Application**

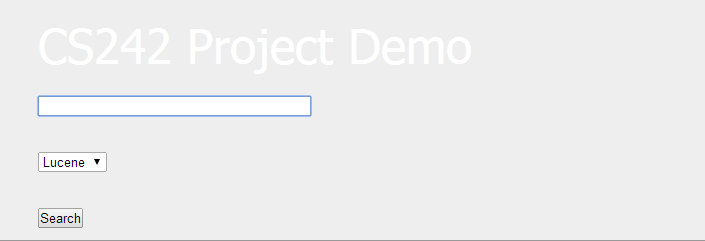
We made use PHP for developing this web Application.

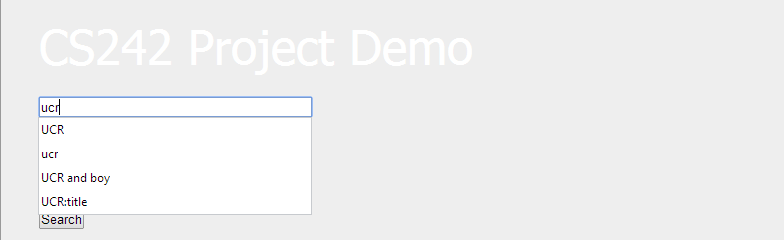
We are making use of the Wamp Server instead of Tomcat because we are not creating any servlets for our Web Application .Therefore running the executable jars with Wamp is much easier in that case.

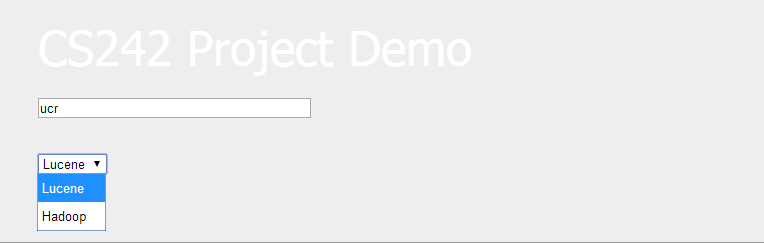
We made use of the explode functionality of PHP in order to parse the results generated by our executable jars and then we display the results to our user.

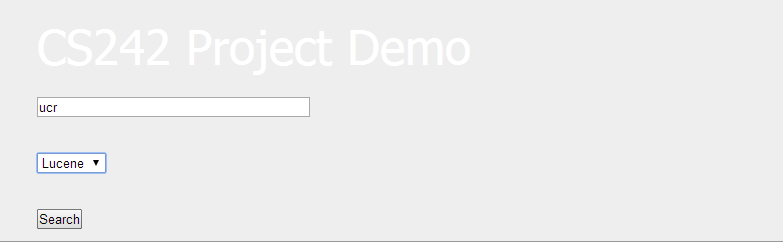
In the following snapshots of our Web Application we give the user a text box to enter the query, user can than select between the Hadoop and Lucene indexes.

**Snapshots of Web Application**-

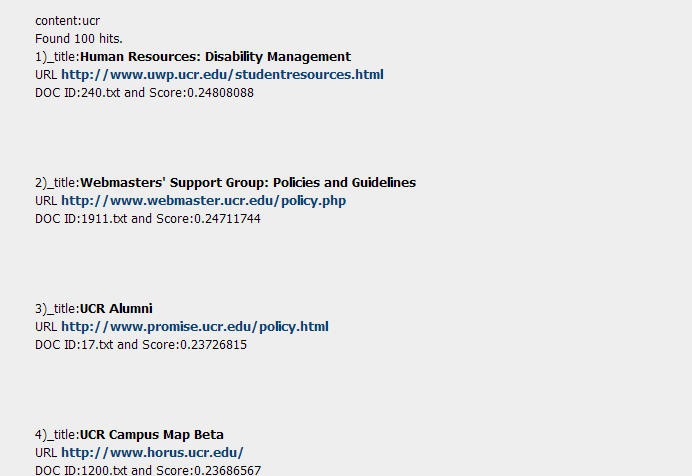




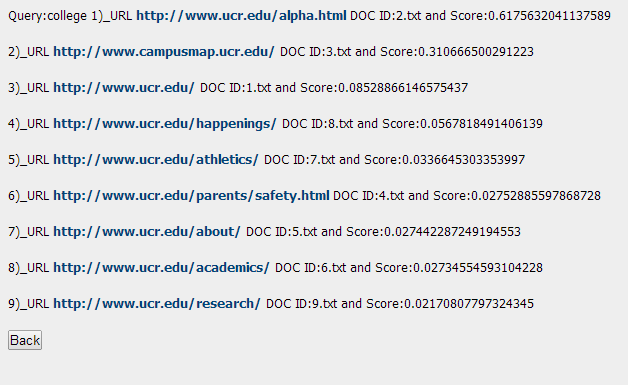




**results of lucene for a query UCR**



**Similarly the results of Hadoop for a query college**



**7> Obstacles and Solutions**

**1.** We first started with small input files for our Hadoop indexing, but the results we were getting were not very efficient, so we decided to merge the input files with size of 100 MB which made our Hadoop processing performance efficient.

**2.** We started with processing the output of Hadoop indexes in one single file, but that wasn’t a very efficient way when we were doing the search. So we decided to split the output files based on the first character of the term. This ensures that when we are doing the search we do not have to read all the index files.

**8> Limitations**

**1.** We haven’t added the snippet generation algorithm for our query results.

**2.** We would like to make our map reduce job more efficient by adding a combiner which would calculate the term frequency as intermediate result and give it to the reducer. This would ensure that we make use of less number of counters in our map reduce jobs.

**9>Instructions on how to run the System**

**Instructions on how to run the Lucene Searcher**

We have provided one lucene\_searcher.bat (Windows) file which in turn calls the Lucene\_Searcher executable jar file.

Example: lucene\_searcher.bat <Query:ucr> <Lucene Index Directory: Lucene\_index>

Here Lucene\_Index is the directory where all lucene indexes are created

Here is a sample working example

lucene\_searcher.bat ucr Lucene\_Index

**Instructions on how to run the Hadoop Searcher**

We have provided one hadoop\_searcher.bat (Windows) file which in turn calls the Hadoop\_searcher executable jar file.

Example: hadoop\_searcher.bat <Query:college> <Hadoop Index Directory: Hadoop\_Output>

Here Hadoop\_Output is the directory where all Hadoop indexes are created

Here is a sample working example

hadoop\_searcher.bat college Hadoop\_Output

**Instructions on how to run the Hadoop indexer**

We have provided one hadoop\_indexer.bat (Windows) file which in turn calls the MapReduce-0.0.1-SNAPSHOT executable jar file.

Example: hadoop\_indexer.bat <Hadoop Input: Hadoop\_Input > <Hadoop Index Directory: Hadoop\_Output>

Here Hadoop\_Input is the directory where all input crawled files are kept

Here Hadoop\_Output is the directory where all Hadoop indexes are created

You can also directly take the MapReduce-0.0.1-SNAPSHOT executable jar file for running on the cluster

You have to create an Input File directory in your HDFS.

Copy the contents from Hadoop\_Input folder to your newly created input folder in HDFS.

Say the input directory you created is In

Sample command-

hadoop jar MapReduce-0.0.1-SNAPSHOT.jar In Out

**Instructions for running the web Application**

1) Install Wamp Server from the following location-

<http://sourceforge.net/projects/wampserver/>

2) Go to the folder wamp, inside there would be an exe named wampmanager , you can start the server with that, check at the bottom right of your screen to check wamp server has got a green light to ensure its running.

3) Copy folder named Web\_Application from my submission and copy it inside the folder named www inside wamp folder.

4) Now go to the browser and type

http://localhost/ Web\_Application /

Now you will be all set to use the web application

Also provided one readme.txt and all the source files