**Introduction:**

A search engine must be able to handle the high demand and it also needs to deliver useful results to the users. Since there are large number of web pages, an efficient data flow for accurate results must be considered. The aim of this project is to create a search engine using the concepts that we have discussed in previous projects.

**Architecture and Implementation:**

The architecture of a search engine depends on effectiveness (quality of results) and efficiency (speed: response time and throughput).

The PageRank algorithm is used to improve the effectiveness of the results. A user may be looking for a particular web page. But there are a lot of web pages that include the phrase, or word a user is looking for. Few may be more significant than others. So, it is necessary to rank and order the result of a search before showing it to the user. PageRank algorithm ranks the web pages that contain the word or phrase a user is looking for based on their relevance. It is an iterative algorithm that prioritizes a web page based on the web pages that link to it and the web pages that it links to. Each web page is presented as a node of a directed graph where the edges of each node are the web pages it links to and the web pages that link to it.

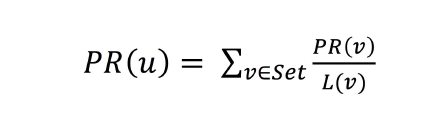


Figure 1: Mathematical PageRank for a simple network from Wikipedia

Figure 1 shows a web graph consisting of 11 vertices A, B, C, D, E, F, G1, G2, G3, G4, G5. Each vertex refers to a unique webpage, and the directed edge means there is one link from the source webpage to the target webpage. The percentage on each vertex represents the rank value of each webpage.

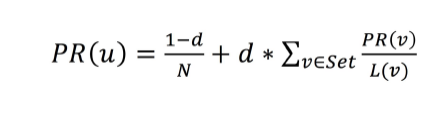
PageRank calculates the probability for a random user to access each web page in the hyperlinked set of webpages. Each iteration of PageRank algorithm calculates a new access probability for each web page based on values calculated in the previous iteration. The process will repeat until the number of current iterations is bigger than predefined maximum iterations, or the Euclidian distance between rank values in two subsequent iterations is less than a predefined threshold that controls the accuracy of the output results.

The equation to calculate PageRank value for a page u is expressed as below:



The vertices v in the above equation contains all the webpages that point to target webpage ‘u’. L(v) refers to the out degree of each webpage in the vertices set. The initial rank values of each webpage, PR’(u) can be any double value and is usually set to 1/N, where N is the total number of urls. After several iterations, the rank values converge to a stationary distribution regardless of what their initial values are.

According to the PageRank theory, a user who is randomly clicking on links will eventually stop clicking at some stage. The probability that the person will continue to click is called the damping factor. Various studies have tested different damping factors. The damping factor d is 0.85 in general. The equation considering damping factor is shown below:



The input to the PageRank algorithm is a file containing data in the form of adjacency matrix of a certain graph. The adjacency matrix that holds the information of each of the web pages may result in a very large file since the web is large. The file may not fit into the memory of a single computer, sometimes it may not fit in the memory of as many as 10,000 computers. So, a distributed system architecture is needed to tackle this problem. One of the ways is to use Hadoop. The data flow of Hadoop implementation to calculate the PageRank values of the urls is represented as below:



Figure 2: Hadoop PageRank

Since the information that reflects the architecture of the web is huge, it must be distributed in several machines for the computation and it must be stored in large databases that facilitates its access. Apache HBase is used for this purpose. It is an open source, distributed, versioned and nonrelational database model for random and real time read/write access to Big Data. We will load the data from an existing HBase records instead of loading data from HDFS.

We use the ClueWeb09 dataset for this project. It consists of about 1 billion webpages in ten languages. The ClueWeb09 dataset is composed of web pages crawled from the Internet. The uploaded table schemas are designed as shown below:



Figure 3: Data table schema for storing the ClueWeb09 dataset

This information will then be used to create an Inverted Index table which has the unique term’s occurrences in all documents from the clueWeb09 dataset. In the clueWeb09IndexTable table each term will have the same structure, with term as rowkey, values contained in documentId, and the occurrence of the term within this document shown.

Each row record of columnfamily “frequencies” is unique, where the rowkey is the unique term stored in byte format, column name is the documentId that contains this term, and value is the term frequency shown per document. Note that each row has multiple columns. The result must be loaded to HBase clueWeb09IndexTable. The purpose of an inverted index is to allow fast full text searches when a document is added to the database. It requires increased processing.

For a given set of documents, each composed of a series of terms (words), it records the information for each term, the subset of documents containing the term in its text. Below is the clueWeb09IndexTable table schema for storing term frequencies and their related documentId.



Figure 4: clueWeb09IndexTable table schema for storing term frequencies and their related documentId

**Experiments**

**Setup**

The Hardware and Software requirements are as follows:

* Hardware: Baremetal nodes.
* Programming language: Java.
* Software environment: Hadoop, MapReduce, HDFS, JRE, HBase.
* Runtime library: Hadoopcore1.1.2, commonscli1.2, commonslogging1.1.1.

**Input and Output**

To build a search engine, we must count with the things we have mentioned before. A database which contains the PageRank values of the available web pages and the Inverted Index table for the words in each of the web pages. With this information, when a user searches for a specific word, it must retrieve the web pages where that word appears and order the result based on the PageRank values they have.

The search engine has three stages in terms of implementation:

1. The construction of the table where all the distinct words across all the urls and their frequencies are stored (data table).

2. The construction of the table where all the words are mapped to a document (inverted index table).

3. The construction of the PageRank table, where all the urls get their respective pageRank value.

We will explain the Hadoop data flow for the construction of these tables moving forward.

**Data Table**

The Mapper is responsible for counting the number of times a word appeared in a given input record of the HBase and then output it to the Reducer. The map function receives the input as <key, value> pair of the form <ImmutableBytesWritable row, Result result> where row is the row of an HBase record related to a specified URI and result is the stored text for the URI. To access the actual content in the HBase table, the following function is used in the code:

**byte[] contentBytes = result.getValue(Constants.CF\_DETAILS\_BYTES,Constants.QUAL\_CONTENT\_BYTES);**

where Constants.CF\_DETAILS\_BYTES indicate the column family “details” and Constants.QUAL\_CONTENT\_BYTES is the column content of the table shown above.

We use the **getWordFreq**() method to get the word and its frequency in HashMap format. The output <key, value> pair of the map function is <Text word, LongWritable freqs> where “word” is the tokenized word extracted from the input row and “freqs” is the frequency of that word returned by the function **getWordFreq**().

The Reducer is responsible to count the final frequency of each word, adding all the partial results from the Map function. This function receives the <key, value> pair in the form <Text word, Iterable<LongWritable> freqs> from the Mapper phase. The reducer counts the value for each word by adding each of its frequencies. The reducer then outputs a **Put** object to add a row in the WordCountTable in the HBase with the count for a specific word. The Put object is filled with the following command:

**WordCountTable.add(Bytes.toBytes("frequencies"), Bytes.toBytes("count"), Bytes.toBytes(totalFreq));**

Figure below shows the clueweb09 WordCountTable schema.

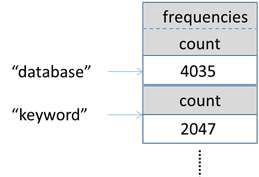


Figure 5: clueWeb09WordCountTable schema

**Inverted Index Table**

We have only the mapper in the HBase FreqIndexBuilder program. We do not have the reducer. HBase FreqIndexBuilder is a “Map-Only” parallel application.

The input to the Mapper is <key,value> pairs of the form <ImmutableBytesWritable rowKey, Result result > where rowKey is the rowkey of an HBase record related to a specified URI and result is the stored text of that URI.

The below call is used to access the content in the HBase table:

**byte[] contentBytes = result.getValue(Constants.CF\_DETAILS\_BYTES,Constants.QUAL\_CONTENT\_BYTES);**

where Constants.CF\_DETAILS\_BYTES indicates the column family “details” and Constants.QUAL\_CONTENT\_BYTES is the column content of the table shown above.

We then make a call to the function getWordFreq() to get the frequency of each word. We store this in the HashMap object named freqs. Once we get the frequencies for each distinct word, a Put object FreqIndexTable is created. We will then add a row to the FreqIndexTable in HBase using the below command:

**FreqIndexTable.add(Bytes.toBytes(Constants.CF\_FREQUENCIES), docIdBytes, Bytes.toBytes(freq));**

where docIdBytes is the rowKey of an HBase record. The output <key,value> pair of this function is <ImmutableBytesWritable word, Writable FreqIndexTable>.

**PageRank Table**

The code involves 3 MapReduce jobs to perform CreatGraph, PageRank and CleanupResult tasks. The input file has the data in adjacency matrix format of a certain graph. The data flow of this process is shown in figure 2. The program works as follows:

1. **CreatGraph:**

The CreateGraph job runs first. This job is used to transform the input file. The input to the map function is a text file with the form: sourceUrl targetUrl1 targetUrl2 targetUrl3 …, The output of the reduce function is a <key, value> pair of the form: <sourceUrl, InitialRank#targetUrl1#targetUrl2#targetUrl3…>.

1. **PageRank:**

The next job is of PageRank. The output of CreateGraph reduce phase is the input for PageRank map phase. The PageRank job calculates the actual PageRank value for each node of the graph. The map function calculates the contribution of sourceUrl to each of the targetUrls it is pointing to for each line of the file. Map function creates two <key, value> pairs. One <key, value> pair is created for each of the targetUrl with the form: <targetUrl(u), rankValue(u)> where rankValue(u) is the InitialRank of the sourceUrl divided by the number of its outlinks. The second <key, value> pair is for the outlinks which has the form: <sourceUrl, targetUrl1#targetUrl2#targetUrl3…>. The reason this is done is that the program needs to run several iterations for the data to converge. The data will be needed for the next iteration.

The Shuffling and Sort phase sorts the map output based on the keys. So, for each urlin the graph, we can have two possible types of values: One being rankValue(v) where v can be 1 to N. N being the total number of nodes that points to node *v*. Second is the urls that node *u* points to: #targetUrl1#targetUrl2#targetUrl3… and so on.

The reduce function of the PageRank phase calculates value for *ith* iteration for every node. The reduce function adds all the rankValues, i.e. PR(*V*)/L(v) for node *u* and compute its actual PageRank value using the following formula:



Where d is damping factor which is equal to 0.85.

This PageRank value is computed in parallel for each node. The values are formatted in the form of: <sourceUrl(u), PR(u)#targetUrl1#targetUrl2#targetUrl3…>. The output of the reduce and the input of the map functions have the same format, to maintain the consistence throughout the iterations.

1. **CleanupResult:**

This phase also includes the sort functionality. The output of the map function of Pagerank phase is the input to this phase. The map phase in CleanupResult cleans the values by removing the outlinks. These values are the formatted in the form of <sourceUrl(u) , PR(u) #targetUrl1 #targetUrl2 #targetUrl3 …> and stored in the HDFS by the CleanupResult job.

**Search Engine**

Once the information needed is gathered by the algorithms explained above, the PageRank data is loaded to an HBase table. In this way, the data flow of a search is explained in figure 6.