Cloud Infrastructure for Large-Scale Search Engine

CSCI – B649 Cloud Computing

Maria Soledad Elli
mselli@iu.edu

Suhas Jagadish
jagadiss@iu.edu

1. Introduction

Given the huge amount of web pages nowadays, a search engine needs to be able to handle the high demand and it also needs to deliver useful results to the users. Because of this, an efficient data flow for accurate results must be taken into account.

Thus, the aim of this project is to present and discuss the methods to create a search engine using the concepts that we have learned in previous class projects.

2. Architecture and Implementation

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

In order to improve the effectiveness of the results, the PageRank algorithm, proposed by Lawrence Page and Sergey Brin in 1998 [1], is used in order to rank the web pages that contain the word or phrase a user is looking for and rank them based on their relevance. While there may be a lot of web pages that include the phrase or word a user is looking for, some of them are more significant than others. As a consequence, it is necessary to rank and order the result of a search before showing it to the user.

Basically, this iterative algorithm gives an idea of how important a web page is based on the web pages that link to it and the web pages 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.

PageRank calculates a numerical value for each element of a hyperlinked set of webpages, which reflects the probability that a random surfer will access that page. An iteration of PageRank calculates the 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.

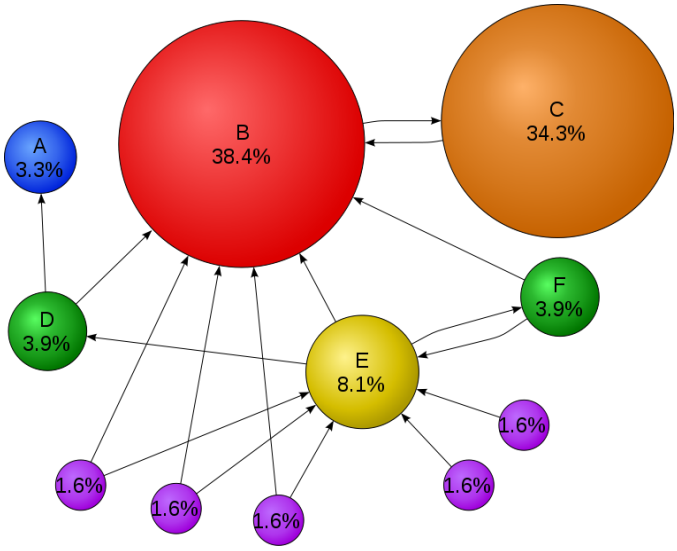

Figure 1. Mathematical PageRank for a simple network, expressed as percentages. (Image from Wikipedia)

In a general case, the PageRank value for any page *u* can be expressed as:

$$PR(u) = \sum_{v \in Set} \frac{PR(v)}{L(v)} \quad \text{Eq.1}$$

The vertices seen in the right of equation 1 contain all the webpages that point to target webpage 'u'. The L(v) refers to the out degree of each webpage in the vertices set. The initial rank values of each webpage, like PR'(u), can be any double value and usually those are set to 1/N, where is the total number of urls. After several iteration calculations, the rank values converge to the stationary distribution regardless of what their initial values are.

The PageRank theory holds that even an imaginary surfer who is randomly clicking on links will eventually stop clicking. The probability, at any step, that the person will continue is a damping factor *d*. Various studies have tested different damping factors, but it is generally assumed that the damping factor will be around 0.85. The formula considering damping factor is shown in equation 2.

$$PR(u) = \frac{1-d}{N} + d * \sum_{v \in Set} \frac{PR(v)}{L(v)} \quad \text{Eq. 2}$$

Since the web is so large, the adjacency matrix that holds the information of each of the web pages doesn't fit into the memory of any single (or even any 10,000) computers. That is why we have to tackle this problem with a distributed system architecture with the help of applications like Hadoop. Figure 2 shows the data flow of the Hadoop implementation for the calculation of the PageRank values of the urls.

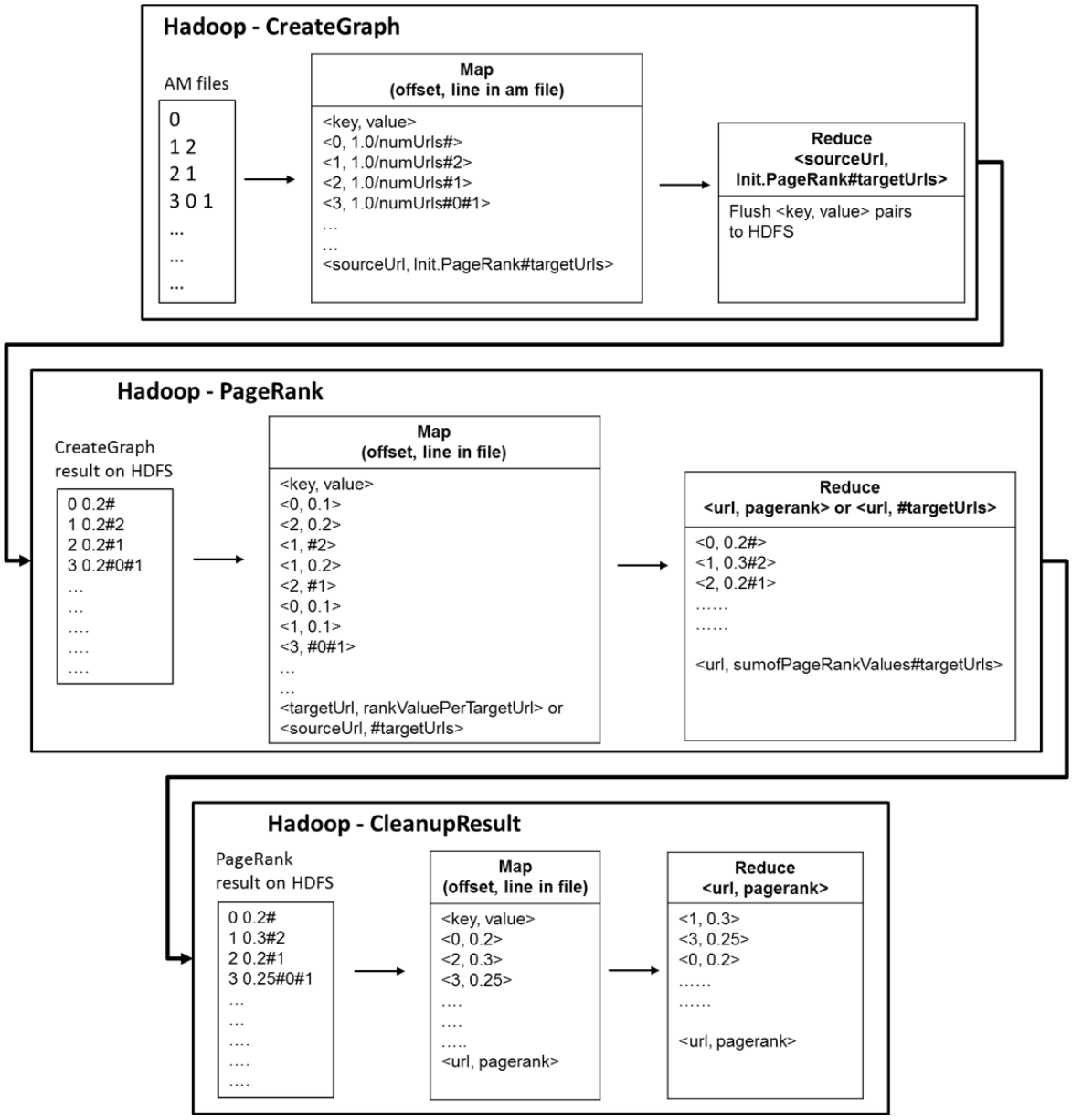

Figure 2. Hadoop PageRank

As mentioned before, the information that reflects the architecture of the web is so huge that must be distributed in several machines for the computation, but also it must be stored in large databases that facilitates its access. This is why we use Apache HBase which is an open-source, distributed, versioned and non-relational database model for random and real time read/write access to Big Data. Thus, instead of loading data from HDFS, we will load the data from an existing HBase records.

Specifically, for this project we used the ClueWeb09 dataset, which was created to support research on information retrieval and related human language technologies. It consists of about 1 billion webpages in ten languages that were collected in January and February 2009. Since the ClueWeb09 dataset is composed of web pages crawled from the Internet, the uploaded table schemas are designed as shown in Figure 3.

Figure 3. Data table schema for storing the ClueWeb09 dataset

This information will then be used in order to create an Inverted Index table which has the unique term's occurrences in all documents from the clueWeb09 dataset. 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, at a cost of increased processing when a document is added to the database. So, for a given set of documents, each composed of a series of terms (words), it records the following information: for each term, which subset of documents contains it in their texts. Figure 4 shows the schema of the clueWeb09 inverted index table.

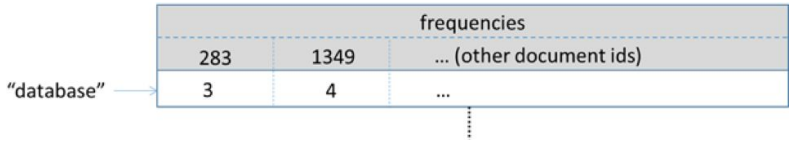

Figure 4. clueWeb09IndexTable table schema

3. Experiments

3.1. Settings

For this project, the Hardware and Software setup is as follows:

- Hardware: Bare-metal nodes.
- Programming language: Java.
- Software environment: Hadoop, MapReduce, HDFS, JRE, HBase.
- Runtime library: Hadoop-core-1.1.2, commons-cli-1.2, commons-logging-1.1.1.

3.2 Input and Output

In order to build a search engine, we have to count with the things we've 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 available, when a user searches for a specific word, it is just a matter of retrieve the web pages where that word appears and order the result based on the PageRank values they have.

As mentioned before, 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 (inver index table).
3. The construction of the PageRank table, where all the urls get their respective pageRank value.

In the next steps, we will explain the Hadoop data flow for the construction of these tables.

Data Table

The Mapper is in charge of 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 as input the <key, value> pair in the form of <ImmutableBytesWritable row, Result result> where *row* is the row of an HBase record related to a specified URI and *result* is the text stored the belongs to that 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 indicates the columnfamily "details" and Constants.QUAL_CONTENT_BYTES is the column content of the table shown in the previous section.

The output <key, value> pair of this 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 in charge of count the final frequency of each particular 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>, which is basically the output of the Mapper phase. After counting the final values for each distinct word, the reducer 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(sum));
```

Figure 5 shows the WordCountTable schema.

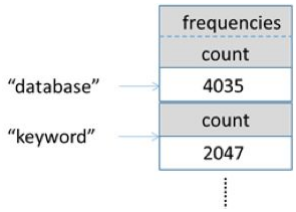

Figure 5. clueWeb09WordCountTable table schema

Inverted Index Table

The The HBase FreqIndexBuilder is a "Map-Only" parallel application wherein we just have the Mapper class and the Main program.

The Map function receives as input the <key,value> pair in the form of <ImmutableBytesWritable rowKey, Result result > where *row* is the rowkey of an HBase record related to a specified URI and *result* is the stored text of that 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 indicates the columnfamily "details" and Constants.QUAL_CONTENT_BYTES is the column content of the table shown in the previous section.

We then create a HashMap object named *freqs* which returns the frequency of each word returned by the function getWordFreq(). Once we get the frequencies for each distinct word, a Put object is created which will add a row in the FreqIndexTable in HBase using the below command:

```
FreqIndexTable.add(Bytes.toBytes("frequencies"), docIdBytes, Bytes.toBytes(freq));
```

where docIdBytes is the rowkey of an HBase record. The output <key,value> pair of this function is <Text word, <docId, frequency>>.

PageRank Table

The code involves 3 MapReduce jobs which performs CreatGraph, PageRank and CleanupResult tasks. The daya flow of this process is presented in figure 2. The algorithm works as follows:

1. CreateGraph job is first used to transform the input file containing the adjacency matrix of a certain graph. The input to the map function for this job is a text file with the form: sourceUrl $targetUrl_1$ $targetUrl_2$ $targetUrl_3$ …, and the output of the reduce function is a <key, value> pair with the form: <sourceUrl, InitialRank#$targetUrl_1$#$targetUrl_2$#$targetUrl_3$…>.

2. After this step, the PageRank job starts to calculate the actual PageRank value for each node of the graph. For each line of the file, the map function calculates the contribution of the sourceUrl to each of the targetUrls it is pointing to. Thus, a <key, value> pair is created for each of the targetUrl with the form: <$targetUrl_i$, $rankValue_i$> where $rankValue_i$ is the InitialRank of the sourceUrl divided by the number of outlinks it has. In addition to this, the map function creates another key-value pair containing all its outlinks: <sourceUrl, #$targetUrl_1$#$targetUrl_2$#$targetUrl_3$…>. This is because the program needs to run several iterations until it converges and in order to do so, the inputs and outputs between the map and reduce tasks need to be consistent.

3. Once the map function is done, the Shuffling and Sort phase sorts the map output based on the keys. So, for each url *i* in the graph, you could have two possible type of values: $rankValue_j$ where *j*:1…N, being N the total number of nodes that points to node *i*, and all the urls that node *i* points to: #$targetUrl_1$#$targetUrl_2$#$targetUrl_3$… and so on.

4. In the reduce function of the second job, the PageRank value for the *ith* iteration is computed for every node. The reduce function adds all the rankValues, PR(*j*)/L(*j*), for node *i* and compute its actual PageRank value with the formula from Equation 2, where d equals to 0.85.

In this way the PageRank value is computed in parallel for each node. These values are then formatted in the form of: <$sourceUrl_i$, PR_i#$targetUrl_1$#$targetUrl_2$#$targetUrl_3$…> and stored in a file on the HDFS by the CleanupResult job. As you can see, the output of the reduce and the input of the map functions have the same format, to maintain the consistence throughout the iterations.

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.

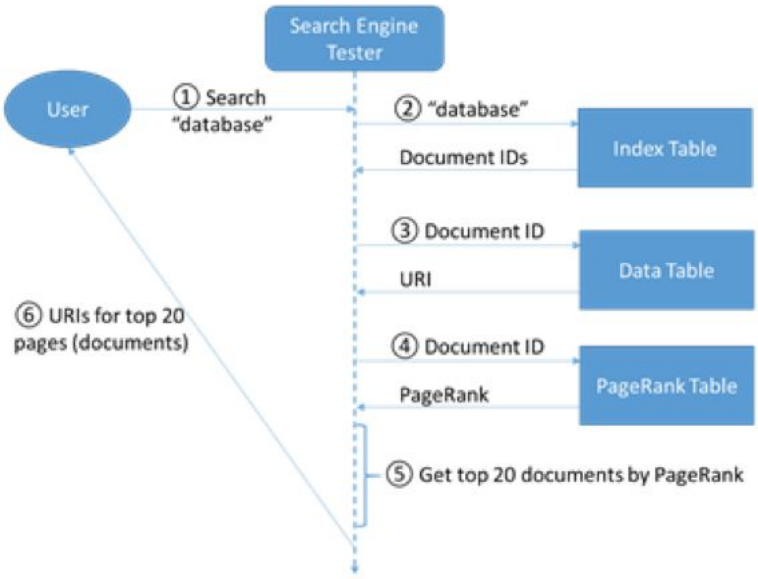

Figure 6. Search Data Flow

Code snippet:

Step 1: get the document ID of one page, as well as the keyword's frequency in that page.

```
// Write your codes for the main part of implementation here
// Step 1: get the document ID of one page, as well as the keyword's frequency in that page
pageDocId=Bytes.toString(kv.getQualifier());
freq=Bytes.toInt(kv.getValue());
```

Step 2: get the URI of the page from clueWeb09DataTable.

```
// Step 2: get the URI of the page from clueWeb09DataTable
Get tempIndexRow = new Get(kv.getQualifier());
tempIndexRow.addColumn(Constants.CF_DETAILS_BYTES, Constants.QUAL_URI_BYTES);
Result docIndexRow = dataTable.get(tempIndexRow);
pageUri = Bytes.toString(docIndexRow.getValue(Constants.CF_DETAILS_BYTES, Constants.QUAL_URI_BYTES));
```

Step 3: get the page rank value of this page from clueWeb09PageRankTable

```
// Step 3: get the page rank value of this page from clueWeb09PageRankTable
tempIndexRow= new Get(kv.getQualifier());
docIndexRow = prTable.get(tempIndexRow);
pageRank = Bytes.toFloat(docIndexRow.list().get(0).getQualifier());
// END of your code
```

4. Hadoop versus Harp

As we've explained in the previous sections, PageRank implementation is done using two different approaches:

- MapReduce, using standard Hadoop MapReduce where the data is stored on HDFS.
- Map-Collective, using Hadoop MapReduce with HARP plugin where the data is stored on HDFS.

We can see that HARP, which is a Map-Collective communication, takes less time to compute page ranks as compared to MapReduce. This is because there are only two I/Os for each Map-Collective operation, i.e., Read from disk and write

final results to disk, while in case of standard Hadoop MapReduce there are four I/Os for each Map-Reduce operation, i.e., read from disk for map task, write intermediate results to disk, read intermediate results from disk, and write final results to disk.

5. Conclusions

As a conclusion for this project, we could say that we've learned the important concepts to build a large-scale system such as search engine. We've discuss the different methods that are being used in classical search engines and how to make them more reliable and fast.

We've explained the concepts under the PageRank algorithm and the Inverted Index method and we've shown how we implemented those using Map Reduce with Hadoop.

It is worth to mention that this search engine architecture is now changing as the people's search expectations are changing. Nowadays, searches involve a more intelligent process that takes into account not only the PageRank values of each url, but also information about the context of each keyword that is being searched. It also involves a knowledge graph that relates all the information about an entity (an specific company, person or situation) that helps search engines to quickly answer accurately the user's questions. Search engines must be able to understand rather than just index knowledge in large databases.

Google, for example, has created a new indexing system to return fresher results to the users, called Caffeine. Figure 7 shows the new indexing system.

"Content on the web is blossoming. It's growing not just in size and numbers but with the advent of video, images, news and real-time updates, the average web page is richer and more complex. In addition, people's expectations for search are higher than they used to be. Searchers want to find the latest relevant content and publishers expect to be found the instant they publish.

...

Caffeine lets us index web pages on an enormous scale. In fact, every second Caffeine processes hundreds of thousands of pages in parallel. If this were a pile of paper it would grow three miles taller every second. ***Caffeine takes up nearly 100 million gigabytes of storage in one database and adds new information at a rate of hundreds of thousands of gigabytes per day****." [2]*

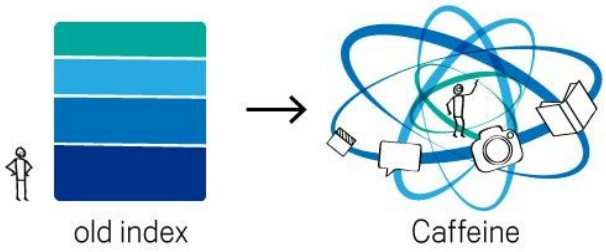

Figure 7. Caffeine, from Google.

This shows us that the more the internet usage grows, the more sophisticated systems have to be. These systems also have to migrate towards a distributed architecture since no large-scale system can fit into one single computer nor can be isolated from the Internet.

6. References

[1] Page, Lawrence, et al. "The PageRank citation ranking: bringing order to the web." (1999)

[2] https://googleblog.blogspot.com/2010/06/our-new-search-index-caffeine.html