Queen Mary University of London

Big Data Processing

Coursework-2 Report

Implementation of a large-scale Search Engine

By

Group J

Raj Kunwar Singh
160950577
Msc. Financial Computing

Muhammad Jawad Akbar
160965869/2
Msc. Big Data Science

Viral Patel
160913675
Msc. Computer Science

Mahfuz Pholby
140745348/2
Bsc. Computer Science

Supervised by

Felix Cuadrado

Contents

1 Literature Survey

The problem required to build a **Searchable Inverted Index** in Mapreduce for the Wikipedia dataset to search for titles in the Wikipedia dataset. Furthermore, the task required to develop the algorithm in a way that took care of massive parallel computing and be more efficient in processing. After building the inverted index, the objective was to then use an algorithm to rank the pages in sequential order by rank using some ranking metric and allow for top results to be displayed.

With massive explosion of data in recent times and people depending more and more on search engines to get all kinds of information they want, it has becoming increasingly difficult for the search engines to produce most relevant data to the users. **PageRank** is one algorithm that has revolutionized the way search engines work. It was developed by Google‘s Larry Page and Sergey Brin. It was developed by Google to rank websites and display them in order of ranking in its search engine results. PageRank is a link analysis algorithm that assigns a weight to each document in a corpus and measures the relative importance within the corpus. However, the MapReduce model is not aware of the inherent random access pattern of web graph which generates lots of network traffic. Thus the efficient processing of large scale Pagerank challenges current MapReduce runtimes. So, **Hadoop MapReduce** framework itself was chosen to implement the large scale search engine for learning purposes and for implementing the basic version of PageRank.

cloud9 libraries (in the form of a jar file) were provided for working with the Wikipedia Dataset and for understanding it better. Getting acquainted with cloud9 libraries gave a better insight into the type of data that was to be dealt with. However, a minimal use of it was made to customize things (as will become clear in the **Implementation** section) according to needs and for learning purposes. The **WikipediaPage**, **XMLFileInputFormat** and some other classes were required while implementing some subtasks of the whole implementation. That is when cloud9 libraries were used and included in the jar files of the corresponding subtasks.

2 Implementation

Implementation of the Large Scale Search Engine consists of 5 sub-tasks which are as follows :

1. **Document Id to Page Title Mapping**
2. **Document Id to Page Contents Mapping**
3. **PageRank**
4. **Building Inverted Index**
5. **Search Terms**

Note : Please copy paste cloud9-2.0.2-SNAPSHOT-fatjar.jar file into the DocIdToTitle/lib and DumpPlainText/lib sub folders. This is done to keep the size of the coursework2 zip file minimal.

2.1 Document Id to Page Title Mapping

This job takes */data/wiki/enwiki-latest-pages-articles.xml* as Input to the Mapper and maps the Document Id to the corresponding Page Title. This job uses the **WikipediaPage** class and hence requires including the **cloud9** jar file in the lib sub folder. This job folder has a different **build.xml** file which compiles all the **.jar** files present in the lib sub folder into a single .jar file.

A Single reducer and **MapFileOutputFormat** is used to generate a single output file to help in random access while listing the search results as we will have the Document Id's and Page Titles will be need for presentability.

Command used :

hadoop-moonshot jar dist/DocIdToTitle.jar edu.umd.cloud9.collection.wikipedia.DocIdToTitle /data/wiki/enwiki-latest-pages-articles.xml coursework2/DocIdToTitle

2.2 Document Id to Page Contents Mapping

This job takes */data/wiki/enwiki-latest-pages-articles.xml* as Input to the Mapper and maps the Document Id to the corresponding Page Contents. This job uses the **WikipediaPage** class and hence requires including the **cloud9** jar file in the lib sub folder. This job folder has a different **build.xml** file which compiles all the **.jar** files present in the lib sub folder into a single .jar file.

The output will be used as input to **Building Inverted Index** sub task for generating the Inverted Index.

Command used :

hadoop-moonshot jar dist/DumpPlainText1.jar edu.umd.cloud9.collection.wikipedia.DumpWikipediaToDocIdPlainText /data/wiki/enwiki-latest-pages-articles.xml coursework2/DocIdPlainText

2.3 PageRank

This subtask further consists of three sub sub tasks which are as follows :

1. **XML Mapper**
2. **Rank Calculate**
3. **Rank Result**

This subtask is handled by **WikiPageRanking.java** which is responsible for sequentially executing the above sub sub tasks to get the Page Rank corresponding to each page.

Command used : A single command is used to handle all the sub sub tasks.

hadoop-moonshot jar dist/PageRank.jar WikiPageRanking

XML Mapper

This job takes */data/wiki/enwiki-latest-pages-articles.xml* as Input to the Mapper and maps the current Document Id to the corresponding Document Ids to which a link from that page exists i.e. This job uses the **XMLInputFormat** class to parse and process XML files. The Reducer outputs, for each page, an assumed initial PageRank i.e. (1.0) and a comma seperated list of links (i.e **Adjacency List**).

The output(*coursework2/PageRank/iter00*)will be used as input to **Rank Result** sub sub task for generating the PageRank.

Rank Calculate

This job takes *coursework2/PageRank/iter00* as Input to the Mapper and maps the current Document Id in three ways. It emits a tuple ⟨ Text, new Text("!") ⟩ to mark the page as existing for each page. It emits the page rank and the number of outgoing links(simply by calculating length of array of links) for each link in the link list of that page. Finally, It also emits the list of links for each page.

The Reducer checks if it is an existing page and then calculates the sum of contribution of each page to which the corresponding page has the link to. Then a new Page Rank is calculated using the formula :

$$PR(p_i) = 1 - d + d * \sum_{p_j \epsilon M(p_i)} \frac{PR(p_j)}{L(p_j)}$$

d is the damping factor chosen as 0.85. $M(p_j)$ denotes the adjacency list of page p_j. $L(p_j)$ denotes the number of outlinks of page p_j.

The newly calculated Page Rank and the comma seperated list of links is outputted for each page. The Output Path is similar to previous sub sub task but for this sub sub task, it depends on the number of iteration. This is handled in **WikiPageRanking.java**.

Rank Result

This job takes the last result path of the previous sub sub task i.e. since we are using 5 iterations. The input path is *coursework2/PageRank/iter05* as Input to the Mapper and maps the current Document Id to the corresponding PageRank.

A Single reducer and **MapFileOutputFormat** is used to generate a single output file at *coursework2/PageRank/result* to help in random access while listing the search results as we will have the Page Rank corresponding to each Document Id to rank the results.

2.4 Building Inverted Index

This job takes *coursework2/DocIdPlainText* as Input to the Mapper and maps each unique word in the page content to the corresponding Document Id. The Reducer creates a **ArrayListWritable** of all the Document Id's corresponding to a term and emits the term and the ArrayListWritable. **ArrayListWritable** is a custom Writable.

A Single reducer and **MapFileOutputFormat** is used to generate a single output file to help in random access while listing the search results as we will have the Document Id's corresponding to a search term.

Command used :

hadoop-moonshot jar dist/InvertedIndex.jar InvertedIndex coursework2/DocIdPlainText coursework2/InvertedIndex

2.5 Search Terms

coursework2/InvertedIndex, *coursework2/PageRank/result* and *coursework2/DocIdToTitle* are taken as Input to this task. These inputs are used to open **Map.Reader** for Map Files for each of the inputs.

The Document Ids corresponding to search term are collected from the **Map.Reader** corresponding to *coursework2/InvertedIndex* and all the PageRanks corresponding to these Document Ids are collected from the **Map.Reader** corresponding to *coursework2/PageRank*. A **PagePair** class is created to represent a Page along with its rank and it implements the **Comparable** class. An ArrayList of PagePair is created containing all the Document Ids(corresponding to the search tem) along with their corresponding PageRanks. This ArrayList is sorted in decreasing order according to PageRank by implementing the **compare** method of the Comparator class.

The Page Titles corresponding to the top 10 Document Ids are collected from the **Map.Reader** corresponding to *coursework2/DocIdToTitle* and the Search Results are displayed on the console.

Command used :

hadoop-moonshot jar dist/SearchTerms.jar SearchTerms coursework2/InvertedIndex coursework2/PageRank/result coursework2/DocIdToTitle Anarchism

where **Anarchism** is the term to be searched.

3 Experimentation and Efficiency with Multiple Terms

Multiple terms can be Searched using the previous command only in the following form

hadoop-moonshot jar dist/SearchTerms.jar SearchTerms coursework2/InvertedIndex coursework2/PageRank/result coursework2/DocIdToTitle The Anarchism

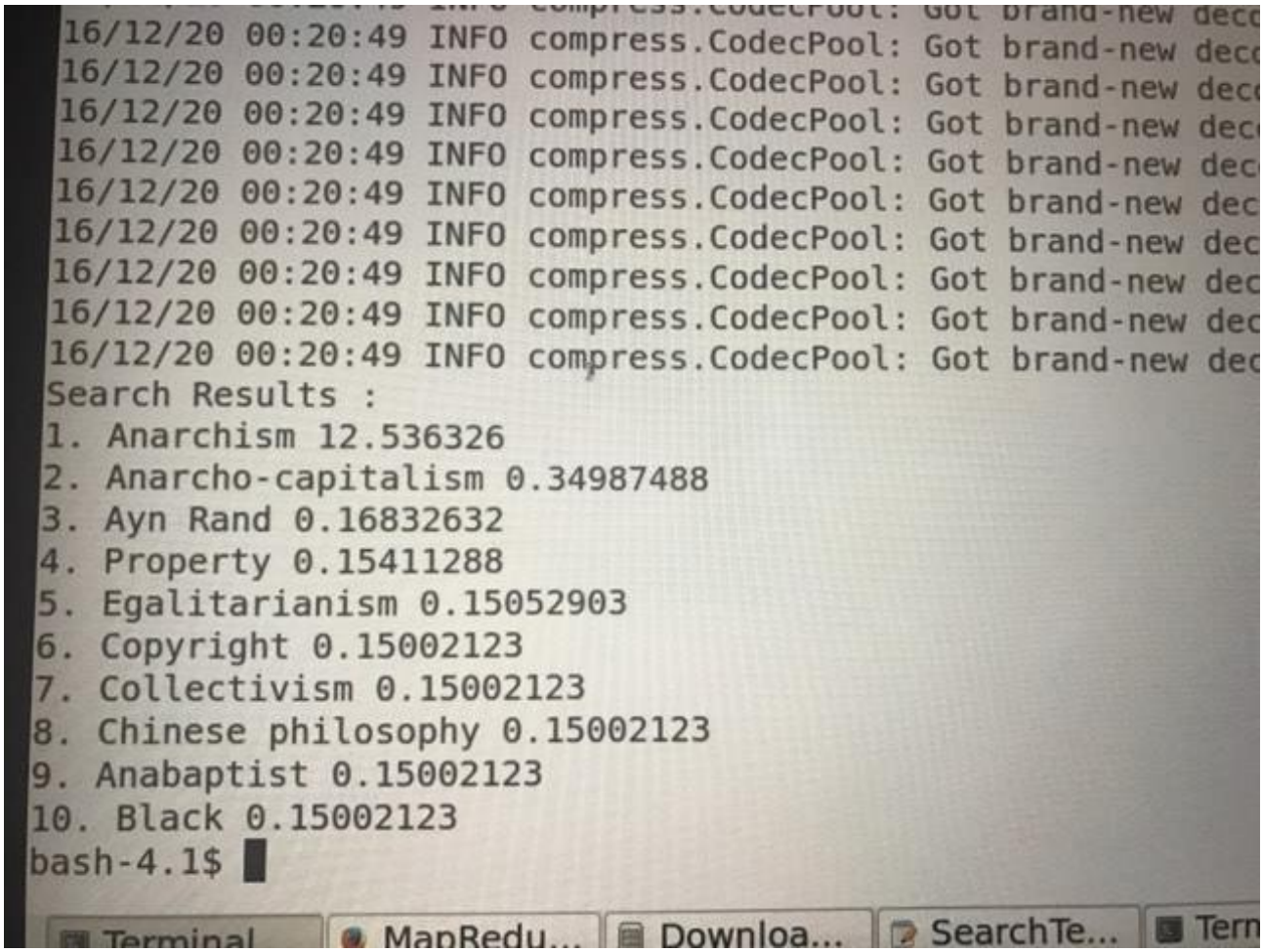
where **The** and **Anarchism** are the terms to be searched.

For efficiency Purposes, a class **TermPair** which implements **Comparable** class is made to represent a term and the number of Document Ids corresponding to it. An ArrayList of TermPair is created containing all the terms along with the corresponding number of Document Ids and then it is sorted in increasing order of number of Document Ids. Then Document Ids corresponding to each term are retrieved sequentially and are intersected with the Document Ids of the next term.

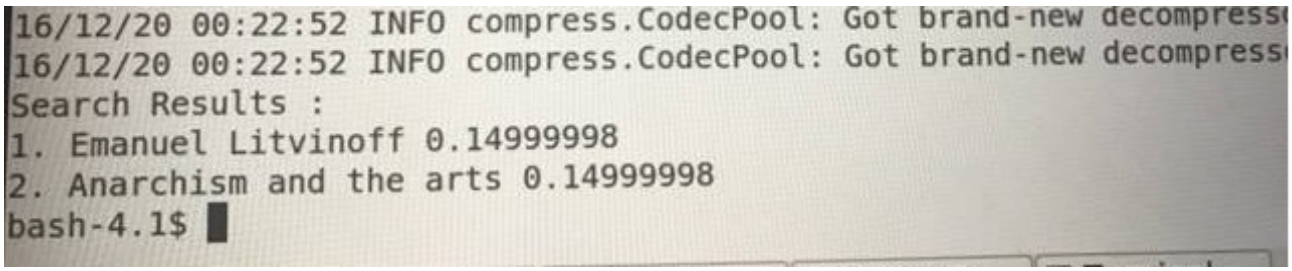
Finally, This Intersection of Document Ids are sorted according to PageRank and the Top 10 results are displayed.

The Efficiency lies in the fact that search terms are sorted according to number of Document Ids and hence the Intersection is performed optimally.

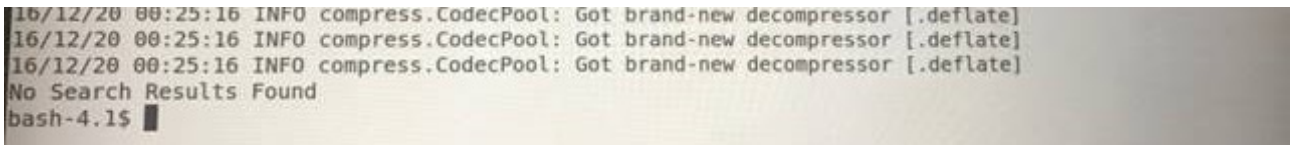
4 Output



Search Results for **The Anarchism**



Search Results for **Anarchism Litvinoff**



Search Results for **Anarchism Rraajj**

5 References

1. http://lintool.github.io/Cloud9/docs/content/wikipedia.html
2. http://salsahpc.indiana.edu/CloudCom2010/EPoster/cloudcom2010_submission_268.pdf
3. https://krex.k-state.edu/dspace/bitstream/handle/2097/17609/AnirudhTadakamala2014.pdf