

Project Report
On
Search Engine Using Hadoop MapReduce



Submitted
In partial fulfilment
For the award of the Degree of

PG-Diploma in Big Data Analytics
(PG-DBDA)

C-DAC, ACTS (Pune)

Guided By:

Mr. Saurabh Panwar

Submitted By:

Gayatri Jawale 230940125019

Ishita Sharma 230940125022

Prajakta Masal 230940125028

Sayali Shedge 230940125046

Srushti Labade 230940125049

Centre for Development of Advanced Computing(C-DAC), ACTS

(Pune- 411008)

Acknowledgement

This is to acknowledge our indebtedness to our Project Guide, **Mr. Saurabh Panwar** C-DAC ACTS, Pune for her constant guidance and helpful suggestion for preparing this project **Search Engine Using Hadoop MapReduce** . We express our deep gratitude towards her for inspiration, personal involvement, constructive criticism that she provided us along with technical guidance during the course of this project.

We take this opportunity to thank Head of the department **Mr. Gaur Sunder** for providing us such a great infrastructure and environment for our overall development.

We express sincere thanks to **Mrs. Namrata Ailawar**, Process Owner, for their kind cooperation and extendible support towards the completion of our project.

It is our great pleasure in expressing sincere and deep gratitude towards **Mrs. Risha P R (Program Head)** and **Mrs. Srujana Bhamidi** (Course Coordinator, PG-DBDA) for their valuable guidance and constant support throughout this work and help to pursue additional studies.

Also, our warm thanks to **C-DAC ACTS Pune**, which provided us this opportunity to carry out, this prestigious Project and enhance our learning in various technical fields.

Gayatri Jawale 230940125019

Ishita Sharma 230940125022

Prajakta Masal 230940125028

Sayali Shedge 230940125046

Srushti Labade 230940125049

ABSTRACT

This project introduces a robust web search engine powered by Hadoop MapReduce, designed to address the scalability and processing challenges. Leveraging the Vector Space Model, documents are represented as sparse vectors using Term Frequency (TF) and Inverse Document Frequency (IDF) metrics, facilitating accurate document indexing and ranking.

The Index Engine comprises components such as Inverse Document Frequency Counter, Term's Frequency Counter, and Document Vectorizer which utilizes MapReduce tasks to compute TF and IDF values.

The Query Vectorizer preprocesses user queries, while the Relevance Analyzer computes relevance scores between queries and documents. The system ensures efficient retrieval of the top N relevant documents through the Content Extractor as the list of ids is provided to the user for selection.

The chosen id is passed to the URL Extractor which fetches the appropriate web page of the id. By showcasing the capabilities of Hadoop MapReduce in handling large-scale web data processing tasks also the URL extractor, this project contributes to advancements in information retrieval technology. The scalable and fault-tolerant architecture enables the search engine to effectively handle the ever-growing volume of online content, paving the way for enhanced search experiences and improved search engine performance in modern web applications.

Table of Contents

S. No	Title	Page No.
	Front Page	I
	Acknowledgement	II
	Abstract	III
	Table of Contents	IV
1	Introduction	01-02
1.1	Introduction	01
1.2	Objectives	02
2	Literature Review	03
3	Methodology/ Techniques	04-09
3.1	Approach and Methodology/ Techniques	04
3.2	Dataset	05
3.3	Model Description	05
4	Implementation	10-16
4.1	Software and Hardware requirements	10
4.2	Workflow	11
4.3	Vector Space Model	14
4.4	Code	15
5	Results	17-18
5.1	Results	17
6	Conclusion	19
6.1	Conclusion	19
7	References	20-21
7.1	References	20

Chapter 1

Introduction

1.1 Introduction

In an era characterized by the exponential growth of online content, the need for efficient mechanisms for information retrieval is more pressing than ever. Web search engines play a pivotal role in facilitating access to relevant information amidst this vast digital landscape. However, the scale and complexity of web data pose significant challenges for traditional search methodologies.

To address these challenges, this project introduces a novel approach to web search engine development leveraging the power of Hadoop MapReduce. Hadoop, a distributed computing framework, offers unparalleled scalability and fault tolerance, making it an ideal choice for processing large volumes of web data. By harnessing the capabilities of Hadoop MapReduce, this project aims to overcome the limitations of traditional search engines and provide a scalable solution for indexing and retrieving web documents.

At the heart of this project lies the Vector Space Model, a mathematical framework for representing documents and queries as vectors in a high-dimensional space. This model enables the computation of Term Frequency (TF) and Inverse Document Frequency (IDF), which are essential for accurate document indexing and ranking. Through a series of MapReduce tasks, the Index Engine processes web documents, computes TF and IDF values, and constructs document vectors, laying the foundation for efficient document retrieval.

The main 3 components of this project are the Index Engine, Ranker Engine and the URL Extractor.

The Index Engine being the first component is responsible for all the indexing related work. The Ranker Engine, built upon the index generated by the Index Engine, analyses user queries and returns the most relevant documents based on their relevance scores. By preprocessing user queries, computing relevance scores, and extracting relevant content, the Ranker Engine ensures the timely delivery of accurate search results to

users. Finally the URL Extractor is provides the desired web page as requested by the user.

In addition to the technical aspects of the project, this report also provides detailed instructions for setting up the search engine environment on a Windows system. From configuring Hadoop and Java to executing indexing and querying tasks, these instructions aim to facilitate seamless deployment and operation of the search engine. Overall, this project represents a significant step forward in web search engine development, offering a scalable and efficient solution for indexing and retrieving web documents. By harnessing the power of Hadoop MapReduce and the Vector Space Model, this project contributes to advancements in information retrieval technology and paves the way for enhanced search experiences in the digital age.

1.2 Objective:

1. Considering multiple documents, this project intends to perform indexing and build search engine so that the top N most relevant documents are returned based on input query analysis.
2. Fetching the appropriate web page as requested by the user.
3. Implement a robust indexing mechanism using Hadoop MapReduce for efficient processing of large web document collections.
4. Provide the best accurate result for the user specified query.
5. Utilize the Vector Space Model to improve document representation and retrieval efficiency.

Chapter 2

LITERATURE REVIEW

Cheng Lin, Ma Yajie et al. [1]. In this research paper, a design and implementation of a vertical search engine is implemented based on Hadoop. As proposed, working of map reduce plays a crucial role in carrying out data processing. In this project the crawler is responsible for gathering the raw data from the internet. The use of indexer is also implemented which converts the documents into a series of index entries, to create inverted index database.

Yu Su et al. [2]. This research paper shows how our indexing support can be used for subsetting operations. Use of Query request generator along with Index Generation along with index retrieval (Retrieval Function) to generate a data reader. The algorithm used here is $\text{Generate Index}(FileName, VarName, Sid)$.

Sukhwant kour Siledar et al. [3]. In this research paper, a brief overview of MapReduce overview and functionality. MapReduce makes use of user specified functions `map()` and `reduce()` to perform its functionality. The former one process a key-value pair and generates set of key-value pairs which are used by later one for merging.

Mahmoud Ahmad Al-Khasawneh et al. [4]. This research paper comprises of a MapReduce Comprehensive Review. This paper stresses the points like Feasibility, Scalability, Efficiency, Fault Tolerance of a MapReduce programming model. Also describes the basics of MapReduce implementations using Hadoop and Cascading.

Zheng-jun Tian et al. [5]. This paper applies the technology of natural language understanding to the website search field, studies document representation theory and realizes the extraction of document eigenvalue using vector space model and TFIDF formula

Chapter 3

Methodology and Techniques

3.1 Methodology:

3.1.1 : Hadoop MapReduce Framework:

The core methodology of this project revolves around leveraging the Hadoop MapReduce framework. Hadoop provides a scalable and fault-tolerant platform for processing large datasets across clusters of commodity hardware. By dividing tasks into smaller sub-tasks and distributing them across multiple nodes, Hadoop enables efficient processing of web documents for indexing and querying.

3.1.2 : Vector Space Model (VSM):

The Vector Space Model is utilized for representing documents and queries as vectors in a high-dimensional space. This model allows for the calculation of Term Frequency (TF) and Inverse Document Frequency (IDF) metrics, which are crucial for document indexing and relevance ranking. TF-IDF values are computed to represent the importance of terms within documents and across the document corpus.

3.1.3 : Indexing Methodology:

The indexing methodology involves three main components: Inverse Document Frequency Counter, Term's Frequency Counter, and Document Vectorizer. These components utilize MapReduce tasks to process web documents, compute TF and IDF values, and construct document vectors. The resulting index is stored in JSON format for efficient retrieval during querying.

3.1.4 : Url Extractor:

The Url Extractor fetches the appropriate web page on the basis of the id provided by the user. The user is supplied with the list of top N ids from which the choice of selection is done. The web page is opened on the existing browser on the user's side.

3.2 Dataset:

The dataset used in this project are text files containing the Wikipedia corpus information in the id, url, title, and text format. 12 text files are present containing the data around 11 to 12 mb. The id field is unique for each word of the dataset. The url field contains direct links to the Wikipedia page of those words. The title field consists a corpus of words from A to Z. The text field contains a description of the words in the title. The text field contains the first Wikipedia page of the link provided in the url field.

3.3 Model Description

1] Index Engine

Index engine consists of 3 parts: Inversed Document Frequency Counter, Token's Frequency Counter and Document Vectorizer

The task of Index Engine is to process all input files and compute vector from defined Vector Space Model for each document. The result is stored as JSON file in form as :

documentID–title:{word1:value1, word2:value2, ..., wordN:valueN}

The title of document is wrapped with document id.

1.1 : Inversed Document Frequency

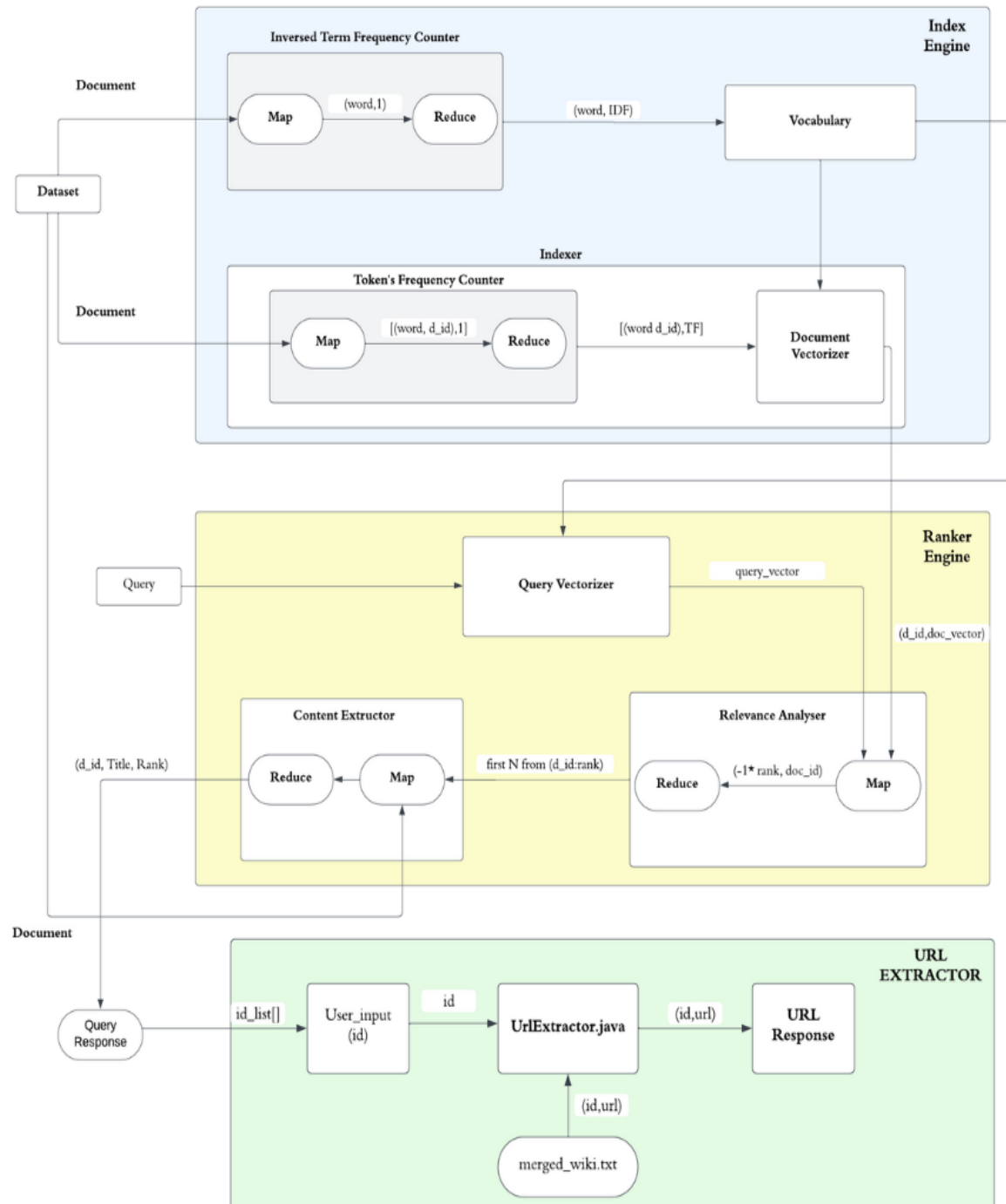
Inversed Document Frequency Counter runs MapReduce task for calculating IDF for each unique word.

The algorithm includes

Map - For each unique word in the document, write pair (key=word, value=1) to the context. Do it for every document in the input path.

Reduce - Sum all values that was written to the context by Map for one word. The result will be exact value of IDF for this word.

The words and associated IDF values then stored in the separate document called "Vocabulary", which will be used further in Index Engine as well as in Ranker Engine



1.2 : Term's Frequency Counter

Term's Frequency Counter runs MapReduce task for calculating TF for each unique word for each document. The algorithm is next:

Map - For each unique word in the document, write pair (key="word documentID", value=1) to the context. Do it for every document in the input path.

Reduce - Sum all values that was written to the context by Map for one key = "word documentID". The result will be exact value of TF for this word in the document with id = documentID.

The keys and associated TF values then stored in the document as intermediate result to be used further in Index Engine.

1.3: Document Vectorizer :

Document Vectorizer runs MapReduce task for producing vector for each document by uniting the results of Inversed Document Frequency Counter and Tokens Frequency Counter. The algorithm is next:

Map - Split each line by TAB symbol and write to context (key=word1, value==word2)

Reduce - Combine all words to one document and save as [*documentID* : *vector*].

2] Ranker Engine

Ranker Engine uses the index produced by Index Engine to analyse query and return top N relevant documents. It consists from 3 parts as well: Query Vectorizer, Relevance Analyser and Content Extractor.

2.1 : Query Vectorizer

Query Vectorizer is the only component that does not use MapReduce. It splits the input query to words and computes TF for each word (basically, the number of occurrences of word in query). This value will be called QTF to avoid confusions. Then it reads words and their IDF from the Vocabulary file produced

by Inversed Document Frequency Counter and based on this information represents query as a vector in the next form:

[word1:QFT1/IDF1, word2:QFT2/IDF2, ...]

2.2 : Relevance Analyzer

Relevance Analyser computes the relevance function between the query and each document using MapReduce. The Relevance function in our case is the inner product (scalar product) of document and query vectors. The algorithm is next:

Map - Compute the relevance function between the query and document. The result will be called the Rank of file. Then, the pair (key = $-1 \times \text{Rank}$, value = document ID) is written to context.

Reduce - Because the Rank value is used as key, document IDs will be already sorted, and all we need is to return pair ($-1 \times \text{key}$, value).

2.3: Content Extractor

Content Extractor receives file with document IDs from Relevance Analyser and cuts the top N lines from it. The title already present in document id in our model.

2.4: Query Response

The id, title and rank is given to the query response where only the list of ids is generated.

3] URL Extractor

3.1 :User Input

The list of ids is provided to the user. This list contains the top N ids of the word searched by the user. The N value is specified in the ranker engine. The ids are arranged in ascending order according to the rank. The 1st id is the best compatible one with the word searched by the user. The user selects an id which suits appropriate to him.

3.2: MergedWiki.txt:

This is a text file which contains all the corpus merged of Wikipedia dataset. This txt file is provided to the UrlExtractor.java for further computation on the data.

3.3: UrlExtractor.java:

This java program takes id as input which is selected by the user along with the mergedwiki.txt file. This program fetches the url of the id as requested from the user taking help from the mergedwiki.txt file.

3.4: URL Response:

The url generated is automatically opened on the users current browser which is the top result of the Wikipedia web page of the word searched by the user.

Chapter 4

Implementation

4.1 Software and Hardware requirements

Operating System :

Linux

Hardware Configuration:

CPU: 8 GB RAM, i5 processor

Software Required:

Use of Hadoop, Java language and Eclipse IDE

- **Hadoop:**

Hadoop 2.7.3 used in this project

Hadoop is an open-source framework for distributed storage and processing of large datasets across clusters of commodity hardware. It provides a scalable, fault-tolerant platform that enables organizations to store, process, and analyze massive volumes of data in a distributed manner.

At its core, Hadoop consists of three main components:

- 1.HDFS
- 2.MapReduce
- 3.YARN

- **MapReduce :**

MapReduce is a programming model and processing framework designed for distributed computing on large datasets across clusters of computers. It simplifies the parallel processing of vast amounts of data by breaking down tasks into smaller, independent units that can be executed in parallel across multiple nodes in a cluster. The MapReduce model consists of two main phases: the Map phase and the Reduce phase.

In our project, MapReduce is used for several tasks related to building a web search engine.

- **Indexing:** MapReduce is utilized to process and analyze a large collection of documents, extracting relevant information and constructing an index for efficient retrieval. In the Map phase, documents are parsed and tokenized, and the frequency of each term within each document is computed. In

the Reduce phase, the intermediate results are aggregated to generate the final index, which maps terms to the documents they appear in.

- **Query Processing:** MapReduce is employed to process user queries and retrieve relevant documents from the index. In the Map phase, the query is tokenized and transformed into a vector representation. In the Reduce phase, the index is queried, and the relevance of each document to the query is computed using similarity measures such as cosine similarity. The top N most relevant documents are then returned as the search results.

- **Java:**

Java 1.8.0_362 version is used in this project.

Java is a widely-used, high-level programming language known for its simplicity, platform independence, and strong community support. It is designed to be object-oriented, meaning it organizes code into reusable components called objects, making it easier to manage and maintain large-scale projects.

In our project, Java is used for several purposes:

1. **Development of MapReduce Jobs:** Java is the primary programming language used to write MapReduce jobs in Hadoop.
2. **Implementation of Data Processing Algorithms:** Java is used to implement various algorithms and data processing tasks required for building a web search engine.
3. **Integration with Hadoop Ecosystem:** Java provides seamless integration with the Hadoop ecosystem, allowing our project to leverage Hadoop's distributed computing capabilities.

- **Eclipse IDE :**

Eclipse is an integrated development environment (IDE) widely used by developers for software development across various programming languages, including Java, C/C++, Python, and more. It provides a comprehensive suite of tools and features to streamline the development process and enhance productivity.

4.2 Workflow

1. Setting up Hadoop and Java in the system.

1.1 Installing Hadoop in a single node cluster

Prerequisites:

1. Java(JDK)
2. Hadoop package

Step1: Verify if java is installed.

java -version

```
talentum@talentum-virtual-machine:~$ java -version
openjdk version "1.8.0_362"
OpenJDK Runtime Environment (build 1.8.0_362-8u372-ga~us1-0ubuntu1~18.04-b09)
OpenJDK 64-Bit Server VM (build 25.362-b09, mixed mode)
talentum@talentum-virtual-machine:~$
```

Step 3: Setting up Hadoop in the system

Step 4: Hadoop Configuration

For Hadoop Configuration we need to modify Six files that are listed below-

1. Core-site.xml
2. Mapred-site.xml
3. Hdfs-site.xml
4. Yarn-site.xml
5. Hadoop-env.cmd

6. Create two folders datanode and namenode

```
sudo mkdir -p /usr/local/hadoop_space
```

```
sudo mkdir -p /usr/local/hadoop_space/hdfs/namenode
```

```
sudo mkdir -p /usr/local/hadoop_space/hdfs/datanode
```

2. Running the Hadoop services.

hdfs namenode -format

```
talentum@talentum-virtual-machine:~$ hdfs namenode -format
24/02/18 17:39:53 INFO namenode.NameNode: STARTUP_MSG:
/*****
STARTUP_MSG: Starting NameNode
STARTUP_MSG:   host = talentum-virtual-machine/127.0.1.1
STARTUP_MSG:   args = [-format]
STARTUP_MSG:   version = 2.7.3
STARTUP_MSG:   classpath = /home/talentum/hadoop/etc/hadoop:/home/talentum/hadoop/share/hadoop/common/lib/commons-digester-1.8.jar:/home/talentum/hadoop/share/hadoop/common/lib/jsr305-3.0.0.jar:/home/talentum/hadoop/share/hadoop/common/lib/curator-recipes-2.7.1.jar:/home/talentum/hadoop/share/hadoop/common/lib/curator-client-2.7.1.jar:/home/talentum/hadoop/share/hadoop/common/lib/jersey-json-1.9.jar:/home/talentum/hadoop/share/hadoop/common/lib/api-asn1-api-1.0.0-M20.jar:/home
```


3.Start yarn services

start-yarn.sh

```
talentum@talentum-virtual-machine:~$ ./run-yarn.sh -s start
starting yarn daemons
starting resourcemanager, logging to /home/talentum/hadoop/logs/yarn-talentum-re
sourcemanaget-talentum-virtual-machine.out
localhost: starting nodemanager, logging to /home/talentum/hadoop/logs/yarn-tale
ntum-nodemanager-talentum-virtual-machine.out
talentum@talentum-virtual-machine:~$
```

4.Jps

```
talentum@talentum-virtual-machine:~$ jps
24816 SparkSubmit
15952 NameNode
29392 Jps
24612 SparkSubmit
28935 ResourceManager
29079 NodeManager
16329 SecondaryNameNode
23068 SparkSubmit
16125 DataNode
26174 Kafka
21631 SparkSubmit
talentum@talentum-virtual-machine:~$
```

5.You have successfully installed *hadoop* on your system. Now to check all you cluster information you can use **localhost:50070** in your browser. The Interface will look like as:

Overview 'localhost:9000' (active)

Started:	Sun Jun 07 14:28:52 +0530 2020
Version:	2.9.0, r756ebc8394e473ac25feac05fa493f6d612ec650
Compiled:	Tue Nov 14 04:45:00 +0530 2017 by arsuresh from branch-2.9.0
Cluster ID:	CID-817fee20-cf62-4852-8b2a-41979f2d439e
Block Pool ID:	BP-1704656119-127.0.1.1-1591520213390

Summary

Security is off.
Safemode is off.
1 files and directories, 0 blocks = 1 total filesystem object(s).
Heap Memory used 72.38 MB of 107 MB Heap Memory. Max Heap Memory is 1000 MB.
Non Heap Memory used 44.15 MB of 47.98 MB Committed Non Heap Memory. Max Non Heap Memory is <unbounded>.

Configured Capacity:	68.4 GB
DFS Used:	24 KB (0%)
Non DFS Used:	10.31 GB
DFS Remaining:	54.58 GB (79.79%)
Block Pool Used:	24 KB (0%)
DataNodes usages% (Min/Median/Max/stdDev):	0.00% / 0.00% / 0.00% / 0.00%
Live Nodes	1 (Decommissioned: 0, in Maintenance: 0)

6. Transfer the data files into Hdfs from local file system.

7. Run the indexer engine.

8. Run the ranker engine.
9. Run the URL Extractor.
10. The requested web page is displayed to the user.

4.3 Vector Space Model

Vector Space Model (VSM) is utilized as the foundation for representing documents and queries in a high-dimensional vector space. The VSM enables efficient retrieval of relevant documents based on their similarity to user queries. Here is a short description of how the VSM is used in this project:

1.Document Vectorization:

Each document in the dataset is vectorized using the TF-IDF (Term Frequency-Inverse Document Frequency) scheme. TF-IDF assigns a weight to each term in the document based on its frequency (TF) in the document and its rarity (IDF) across the entire document collection. The TF-IDF values for all terms in a document are combined into a vector representation, where each dimension corresponds to a unique term in the vocabulary.

2.Query Vectorization:

Similarly, user queries are also vectorized using the TF-IDF scheme. The query vector represents the query's term frequencies adjusted by the inverse document frequency of each term, capturing the importance of terms in the query.

3.Document-Query Similarity Calculation:

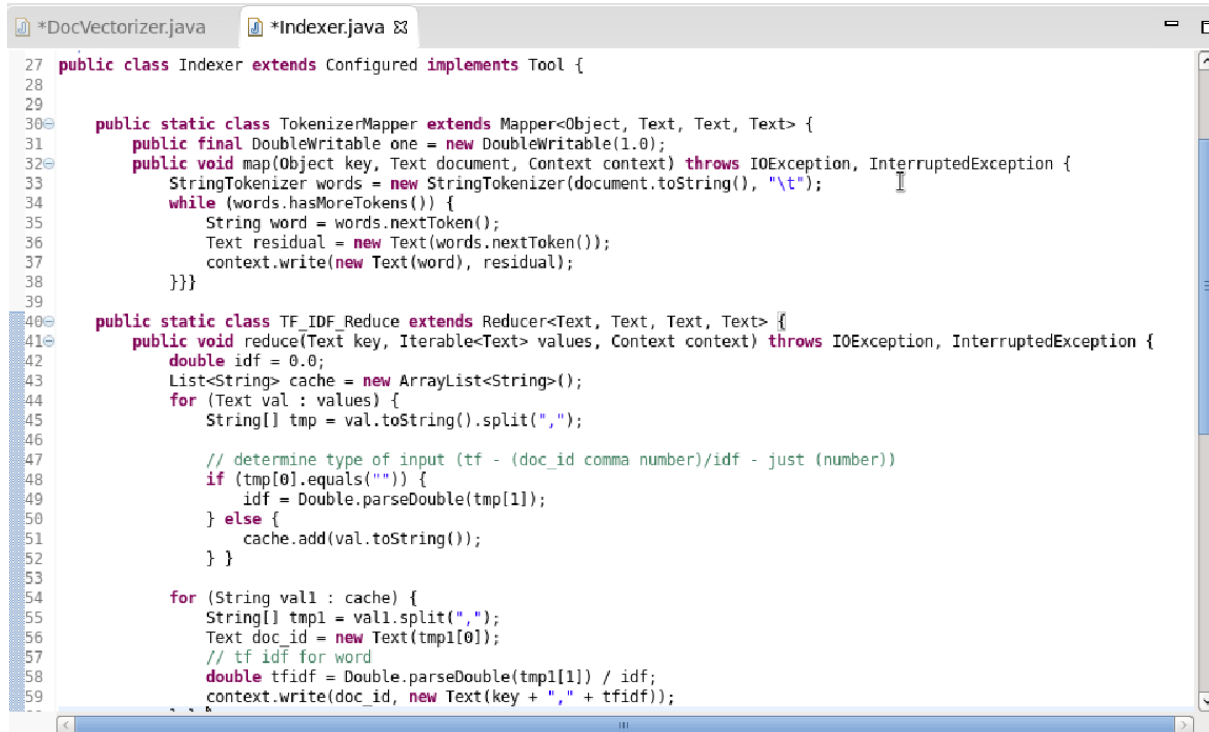
To retrieve relevant documents, the cosine similarity between the query vector and each document vector is computed. Cosine similarity measures the cosine of the angle between two vectors and ranges from -1 (completely dissimilar) to 1 (identical). Documents with higher cosine similarity scores are considered more relevant to the query.

4.Ranking and Retrieval:

The documents are ranked based on their cosine similarity scores with the query vector. The top-ranked documents are retrieved and presented to the user as search results.

4.4 Code

1] Indexer.java

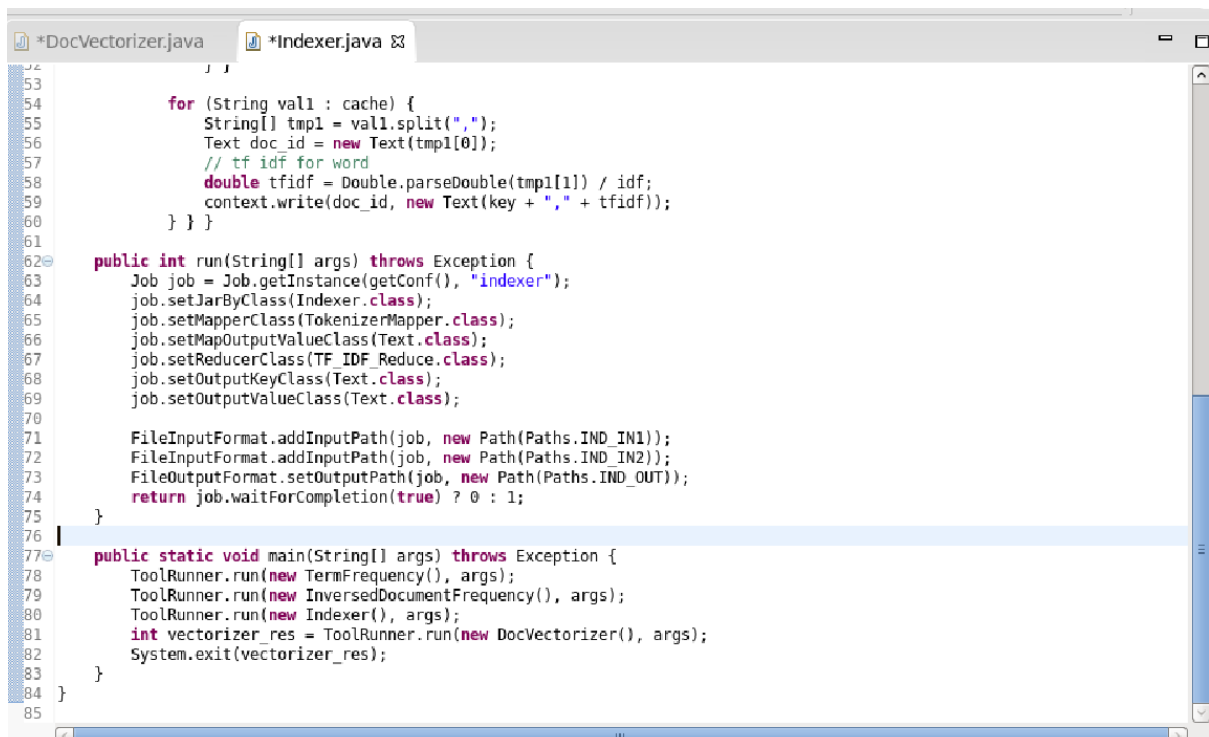


```

27 public class Indexer extends Configured implements Tool {
28
29
30     public static class TokenizerMapper extends Mapper<Object, Text, Text, Text> {
31         public final DoubleWritable one = new DoubleWritable(1.0);
32         public void map(Object key, Text document, Context context) throws IOException, InterruptedException {
33             StringTokenizer words = new StringTokenizer(document.toString(), "\\t");
34             while (words.hasMoreTokens()) {
35                 String word = words.nextToken();
36                 Text residual = new Text(words.nextToken());
37                 context.write(new Text(word), residual);
38             }
39         }
40
41     public static class TF_IDF_Reduce extends Reducer<Text, Text, Text, Text> {
42         public void reduce(Text key, Iterable<Text> values, Context context) throws IOException, InterruptedException {
43             double idf = 0.0;
44             List<String> cache = new ArrayList<String>();
45             for (Text val : values) {
46                 String[] tmp = val.toString().split(",");
47                 // determine type of input (tf - (doc_id comma number)/idf - just (number))
48                 if (tmp[0].equals("")) {
49                     idf = Double.parseDouble(tmp[1]);
50                 } else {
51                     cache.add(val.toString());
52                 }
53             }
54             for (String val1 : cache) {
55                 String[] tmp1 = val1.split(",");
56                 Text doc_id = new Text(tmp1[0]);
57                 // tf idf for word
58                 double tfidf = Double.parseDouble(tmp1[1]) / idf;
59                 context.write(doc_id, new Text(key + "," + tfidf));
60             }
61         }
62     }
63
64     public int run(String[] args) throws Exception {
65         Job job = Job.getInstance(getConf(), "indexer");
66         job.setJarByClass(Indexer.class);
67         job.setMapperClass(TokenizerMapper.class);
68         job.setMapOutputValueClass(Text.class);
69         job.setReducerClass(TF_IDF_Reduce.class);
70         job.setOutputKeyClass(Text.class);
71         job.setOutputValueClass(Text.class);
72
73         FileInputFormat.addInputPath(job, new Path(Paths.IND_IN1));
74         FileInputFormat.addInputPath(job, new Path(Paths.IND_IN2));
75         FileOutputFormat.setOutputPath(job, new Path(Paths.IND_OUT));
76         return job.waitForCompletion(true) ? 0 : 1;
77     }
78
79     public static void main(String[] args) throws Exception {
80         ToolRunner.run(new TermFrequency(), args);
81         ToolRunner.run(new InversedDocumentFrequency(), args);
82         ToolRunner.run(new Indexer(), args);
83         int vectorizer_res = ToolRunner.run(new DocVectorizer(), args);
84         System.exit(vectorizer_res);
85     }
86 }

```

Fig 4.4.1.1: Indexer.java_1



```

53
54
55     for (String val1 : cache) {
56         String[] tmp1 = val1.split(",");
57         Text doc_id = new Text(tmp1[0]);
58         // tf idf for word
59         double tfidf = Double.parseDouble(tmp1[1]) / idf;
60         context.write(doc_id, new Text(key + "," + tfidf));
61     }
62 }
63
64 public int run(String[] args) throws Exception {
65     Job job = Job.getInstance(getConf(), "indexer");
66     job.setJarByClass(Indexer.class);
67     job.setMapperClass(TokenizerMapper.class);
68     job.setMapOutputValueClass(Text.class);
69     job.setReducerClass(TF_IDF_Reduce.class);
70     job.setOutputKeyClass(Text.class);
71     job.setOutputValueClass(Text.class);
72
73     FileInputFormat.addInputPath(job, new Path(Paths.IND_IN1));
74     FileInputFormat.addInputPath(job, new Path(Paths.IND_IN2));
75     FileOutputFormat.setOutputPath(job, new Path(Paths.IND_OUT));
76     return job.waitForCompletion(true) ? 0 : 1;
77 }
78
79 public static void main(String[] args) throws Exception {
80     ToolRunner.run(new TermFrequency(), args);
81     ToolRunner.run(new InversedDocumentFrequency(), args);
82     ToolRunner.run(new Indexer(), args);
83     int vectorizer_res = ToolRunner.run(new DocVectorizer(), args);
84     System.exit(vectorizer_res);
85 }
86 }

```

Fig 4.4.1.2: Indexer.java_2

2] DocumentVectorizer.java

```

*DocVectorizer.java
22 public class DocVectorizer extends Configured implements Tool {
23     public static class TokenizerMapper extends Mapper<Object, Text, Text, Text> {
24         public final DoubleWritable one = new DoubleWritable(1.0);
25         public void map(Object key, Text document, Context context) throws IOException, InterruptedException {
26             StringTokenizer words = new StringTokenizer(document.toString(), "\\t");
27             while (words.hasMoreTokens()) {
28                 String word = words.nextToken();
29                 Text residual = new Text(words.nextToken());
30                 context.write(new Text(word), residual);
31             }
32     }
33     public static class Vectorizer_Reduce extends Reducer<Text, Text, Text, Text> {
34         public void reduce(Text key, Iterable<Text> values, Context context) throws IOException, InterruptedException {
35             JSONObject json = new JSONObject();
36             for (Text val : values) {
37                 String[] tmp = val.toString().split(",");
38                 json.put(tmp[0], tmp[1]);
39             }
40             context.write(key, new Text(json.toString()));
41         }
42     }
43     public int run(String[] args) throws Exception {
44         Job job = Job.getInstance(getConf(), "indexer");
45         job.setJarByClass(DocVectorizer.class);
46         job.setMapperClass(TokenizerMapper.class);
47         job.setMapOutputValueClass(Text.class);
48         job.setReducerClass(Vectorizer_Reduce.class);
49         job.setOutputKeyClass(Text.class);
50         job.setOutputValueClass(Text.class);
51         FileInputFormat.addInputPath(job, new Path(Paths.VECT_IN));
52         FileOutputFormat.setOutputPath(job, new Path(Paths.VECT_OUT));
53         return job.waitForCompletion(true) ? 0 : 1;
54     }
}

```

Fig 4.4.2: DocVectorizer.java

3] Main.java

```

*DocVectorizer.java  *Indexer.java  Main.java
1 package SearchEngine;
2
3 import org.apache.hadoop.util.ToolRunner;
4
5 public class Main {
6     public static final String help = "Wrong arguments, help (arg1 arg2): \n ex1. Indexer /path/to input \n ex2. Query N 'q'";
7     public static void main(String[] args) throws Exception {
8         if (args[0].equals("Indexer")) {
9             if (args.length < 2) {
10                 System.out.println(help);
11                 System.exit(-1);
12             }
13             ToolRunner.run(new TermFrequency(), args);
14             ToolRunner.run(new InversedDocumentFrequency(), args);
15             ToolRunner.run(new Indexer(), args);
16             int vectorizer_res = ToolRunner.run(new DocVectorizer(), args);
17             System.exit(vectorizer_res);
18         }
19         else if (args[0].equals("Query")) {
20             if (args.length < 3) {
21                 System.out.println(help);
22                 System.exit(-1);
23             }
24             int resultOfJob = ToolRunner.run(new RelevanceAnalyzer(), args);
25             System.exit(resultOfJob);
26         }
27         else {
28             System.out.println(help);
29             System.exit(-1);
30         }
31     }
32 }
33
34 }
35
36

```

Fig 4.4.3Main.java

Chapter 5

Results

1] The user is required to provide the query that he wants to search along with the number of top results (N) for the entered query.

```

talentun@talentun-virtual-machine: ~/shared/Search_Engine
talentun@talentun-virtual-machine:~/shared/Search_Engine$ vim hooray1.sh
talentun@talentun-virtual-machine:~/shared/Search_Engine$ ./hooray1.sh
Message: Initializing the Indexer Engine.....
Indexer has completed its execution
talentun@talentun-virtual-machine:~/shared/Search_Engine$
talentun@talentun-virtual-machine:~/shared/Search_Engine$

```

2] The Indexer Engine and Ranker Engine work to provide the top N searches for the user entered query. The user then has to enter the id from the displayed list of ids to view the webpage of the searched query.

```

talentun@talentun-virtual-machine:~/shared/Search_Engine$ vim hooray2.sh
talentun@talentun-virtual-machine:~/shared/Search_Engine$ ./hooray2.sh
*****
WELCOME TO COSMIC CRUISE!
EXPLORE, SEARCH, SOAR!
*****
Enter Your Query:
What does Anarchism mean??
How Many Results Do You Want?
3
Fetching results for What does Anarchism mean??. ....
ID | Title | Rank
12 Anarchism 10.666890640637005
339 Ayn Rand 0.3334096272786458
13299 History of Spain 0.1111721462673611
Enter the id to find corresponding url :
12
URL for object with ID 12: https://en.wikipedia.org/wiki?curid=12
talentun@talentun-virtual-machine:~/shared/Search_Engine$

```

3] The user entered id will be an input to the URL Extractor which will fetch the relevant web page on to the default browser of the user.



Chapter 6

Conclusion

6.1 Conclusion:

In conclusion, this project has successfully demonstrated the feasibility and effectiveness of developing a web search engine using Hadoop MapReduce. Leveraging Hadoop's scalability and fault tolerance, coupled with the Vector Space Model, the system efficiently indexes and retrieves web documents. Through indexing and querying components, the project processes large-scale web datasets, computes relevance scores, and retrieves the most relevant documents. Hadoop's fault tolerance ensures reliable operation, enhancing system robustness. The URL Extractor helps to fetch the desired webpage through id which is the ultimate goal of this project. Overall, this project advances information retrieval technology by leveraging Hadoop for scalable web data processing. Future work could focus on optimizing algorithms and enhancing user interfaces for further improvements.

6.2 Future Enhancement :

1. Integration of Machine Learning Techniques: Incorporating machine learning techniques, such as natural language processing (NLP) and deep learning, could enhance the search engine's capabilities.
2. Semantic Search: Implementing semantic search capabilities would enable the search engine to understand the meaning and context of user queries.
3. Real-time Indexing and Querying: Enhancing the search engine to support real-time indexing and querying would enable users to access the most up-to-date information instantly.
4. Improvement of Relevance Ranking Algorithms: Continuously refining and optimizing the relevance ranking algorithms could improve the accuracy and effectiveness of search results.

Chapter 7

References

7.1 Research Papers

[1] Cheng Lin, Ma Yajie College of Information Science and Engineering, Wuhan University of Science and Technology, Wuhan, Hubei, 430081, China

“Design and Implementation of Vertical Search Engine Based on Hadoop”

[2] Yu Su Computer Science and Engineering The Ohio State University Columbus, U.S.A, Gagan Agrawal Computer Science and Engineering The Ohio State University Columbus, U.S.A, Jonathan Woodring CCS-7 Applied Computer Science Group Los Alamos National Laboratory Los Alamos, U.S.A

“Indexing and Parallel Query Processing Support for Visualizing Climate Datasets”

[3] Sukhwant kour Siledar, Bhagyashree Deogaonkar, Nutan Panpatte, Dr. Jayshri Pagare M.Tech, Department of Computer Science and Engineering, Jawaharlal Nehru Engineering College, Aurangabad, India

“Map Reduce Overview and Functionality”

[4] Mahmoud Ahmad Al-Khasawneh, Siti Mariyam Shamsuddin, Shafaatunnur Hasan, Adamu Abu Bakar,

“MapReduce a Comprehensive Review”

[5] Zheng-jun Tian, Hong-yan Zhang School of Software Henan Institute of Engineering ZhengZhou, China “Optimization Design of Website Search Based on Vector Space Model”

7.2 Links

<https://www.geeksforgeeks.org/how-to-install-single-node-cluster-hadoop-on-windows/>

<https://ieeexplore.ieee.org/document/7938907>

<https://ieeexplore.ieee.org/document/7488534>

<https://ieeexplore.ieee.org/document/6965006>

<https://ieeexplore.ieee.org/document/6337586>