

Processing big data using MapReduce, HDFS, Spark and Hadoop framework

Anh Minh Nguyen
103178955

Abstraction

This article discusses how to process large amounts of data using MapReduce, HDFS, and Hadoop. The article will first provide an overview of HDFS, which uses NameNode and DataNode for large-scale data sets. Secondly, MapReduce will be introduced with two main functions which are the map and reduce functions with two types of daemon nodes including JobTracker and TaskTracker. In this paper, a counting top 25 rated movies program will be implemented in two models in the Hadoop ecosystem including MapReduce and Spark. In order to compare these two models, some criteria will be considered to assess to identify which framework would be better. The project will discuss the result of the implementation of both programs using MapReduce and Spark respectively, as well as provide some insights into the data to demonstrate the comparison between these frameworks.

Table of contents

1.0 Introduction	4
1.1 Background	4
1.2 Purpose	4
2.0 Hadoop concept	4
2.1 HDFS	5
2.1.1 NameNode	5
2.1.2 DataNode	5
2.1.3 Second NameNode	6
2.1.4 Process of HDFS	6
2.2 MapReduce	7
2.2.1 JobTracker	8
2.2.2 TaskTracker	8
3.0 Method	8
4.1 Push input file to HDFS	9
4.2 Map phase	10
4.3 Shuffle and sort phase	11
4.4 Reduce phase	11
4.5 Finding the top 25 most-rated movies	12
4.6 Config assignment job in MapReduce	13
4.7 Apache Spark	13
4.8 PySpark program	13
4.9 Finding the top 25 most-rated movies by PySpark	15
5.0 Result	16
5.1 Result of implementation of Spark and MapReduce	16
5.2 Spark vs MapReduce	18
5.3 Quantitative result	19
6.0 Discussion	19
7.0 Conclusion	19
Reference	20
Appendix	21

List of Figures

Figure 1: A tree showing the functions inside the Hadoop ecosystem	4
Figure 2: The read file process in HDFS	6
Figure 3: Write files process in HDFS.....	7
Figure 4: Interaction between JobTracker and TaskTracker	8
Figure 5:Successful creating a new folder on the terminal.....	9
Figure 6:Successful creating a new folder in on the NameNode monitor web browser.....	9
Figure 7: Successful uploading of a file to the HDFS on the terminal	9
Figure 8:Successful uploading a file to the HDFS on the NameNode monitor web browser .	10
Figure 9: The first 9 lines of ratings.dat.....	10
Figure 10: First 10 lines of the output.....	12
Figure 11: First 10 lines of the output.....	13
Figure 12:First 20 lines of the table after converting data in Spark.....	14
Figure 13: First 20 lines of the table after querying data in Spark	15
Figure 14:First 20 lines of the final output in Spark.....	16
Figure 15: The succeeded jobs on the resource manager monitor web browser	16
Figure 16: The succeeded output files created by MapReduce on the NameNode monitor web browser.....	17
Figure 17: The executed jobs run on PySpark Shell monitor web browser.....	17
Figure 18: The event timeline for executed jobs run on PySpark Shell monitor web browser	17
Figure 19: The succeeded output files created by Spark on the NameNode monitor web browser	17
Figure 20: The final output after execution both of MapReduce and PySpark.	18

1.0 Introduction

1.1 Background

The growth of information technology has considerably increased the volume of data in the 4.0 age, necessitating the processing and analysis of vast and complicated data. The traditional methods and procedures used to handle these data have become incapable of processing massive data quantities in manageable amounts of time. As a consequence, this study would suggest Apache Hadoop, an open-source version of the MapReduce system, as a load-balancing, scalable, highly available application. In addition, the MapReduce system is a well-known data-parallel processing architecture that has been extensively used for processing enormous amounts of data.

1.2 Purpose

This article will describe how Apache Hadoop uses the Hadoop distributed file system (HDFS) to write and read data while utilizing a cluster and the master-slave architecture. The paper also proposes MapReduce, a programming model and implementation that is adaptable enough to handle a variety of real-world applications for handling and producing large datasets. In particular, the MapReduce implementation technique, which is the method for handling enormous amounts of data, will be demonstrated. Additionally, there is a section to compare the performance and scalability of two frameworks in the Hadoop ecosystem which are MapReduce and Spark by implementing a small program.

2.0 Hadoop concept

Hadoop is an open-source Java-based framework that implements the MapReduce engine with heavy support from HDFS (Hadoop distributed file system). The clusters inside Hadoop can store data and perform parallel computation across a large computer cluster. This allows Hadoop to process large-scale datasets and improve fault tolerance ability. Two main components in Hadoop are HDFS (Hadoop distributed file system) and MapReduce. The first framework inside Hadoop, HDFS can split data into multiple smaller blocks and replicate as well as distribute each part across multiple data nodes in the cluster.

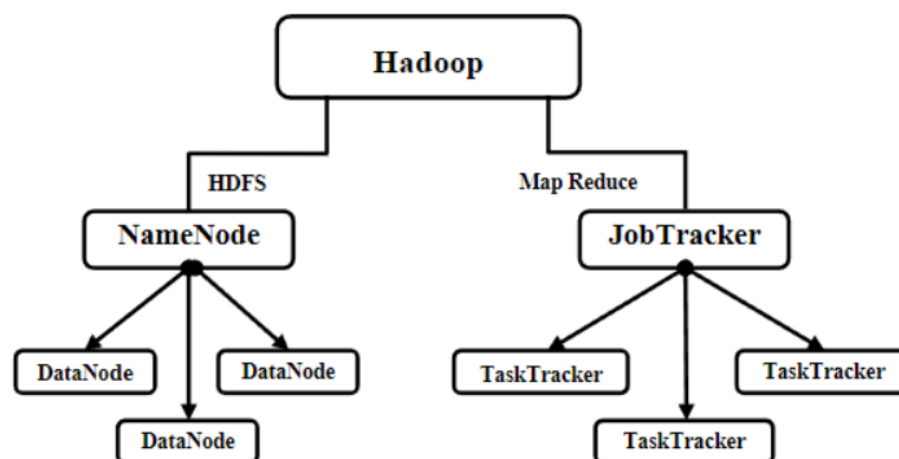


Figure 1: A tree showing the functions inside the Hadoop ecosystem

2.1 HDFS

The idea behind HDFS is to manage data by using master-slave architecture running on one big cluster having multiple nodes to handle big data files. Specifically, the big data file sent to HDFS will be divided into multiple blocks and these blocks are distributed across the nodes on the cluster. This would guarantee the availability and fault tolerance characteristic of HDFS. In Hadoop distributed file system, there are two types of nodes the NameNode acts as the master node, and the DataNode acts as multiple slave nodes.

In HDFS, every received file would be divided into multiple blocks with unique blockID to identify. All the blocks inside one file (except the last block) will have a similar size and this size is called block size which is default is 64MB. However, it can be configured by developers to change the block size, and the number of blocks will depend on the block size as well as the size of the file according to the following formula:

$$numberOfBlock = \lceil fileSize \div blockSize \rceil$$

One of the fundamental feature from the Hadoop distributed file system is the replication of blocks after dividing the file into multiple blocks. This will be done by the NameNode and the DataNode that replicate the block 3 times and store its replica in different DataNode. This functionality of HDFS would consume larger storage. However, it is used to make the data high availability and fault-tolerant. It is possible for a DataNode to be down or unhealthy state, the NameNode would notice the situation and announce other DataNodes to replicate the block inside the inactive DataNode and store them in their storage.

2.1.1 NameNode

NameNode represents the master node, which is responsible for storing and managing metadata about the file system. It manages the file system namespace to manage files and directories, including implementing operations such as opening, closing, and renaming files and directories. Internally, the file is split into multiple blocks then these blocks are replicated and distributed across DataNodes. The NameNode is responsible to identify the mapping of the block and its replica to DataNodes corresponding to a unique blockID. The NameNode would also be in charge of receiving the request from the client application and instructing the DataNode to perform block creation, deletion, and replication. Particularly, the NameNode receives from each DataNode a Heartbeat and a Block report. A Heartbeat indicates the status of DataNode in case it is broken, while the Block report has a list of all the blocks on that DataNode. In terms of, replication and fault tolerance, when a node is down, the NameNode will notice that and force DataNode in the cluster to replicate the data in the DataNode which is broken. That functionality of NameNode guarantees the availability of data as well as the fault-tolerance characteristic of HDFS.

2.1.2 DataNode

A Hadoop cluster would have several DataNodes functioning as slave nodes in the HDFS architecture. These slave nodes play a crucial role in storing data in form of blocks and executing services such as reading and writing on files stored on HDFS. These requests made by the client to read and write the filesystem get processed directly by the DataNodes. To offer high availability and dependability, blocks stored in DataNodes are duplicated following the configuration. To enable quick calculation, the duplicated blocks are dispersed throughout the cluster.

2.1.3 Second NameNode

It is the Secondary NameNode's responsibility to regularly read the file system, report any changes, and then apply those changes to the "fsimage" file. By doing this, NameNode may be updated to boot up more quickly in the future.

2.1.4 Process of HDFS

2.1.4.1 Reading files on HDFS

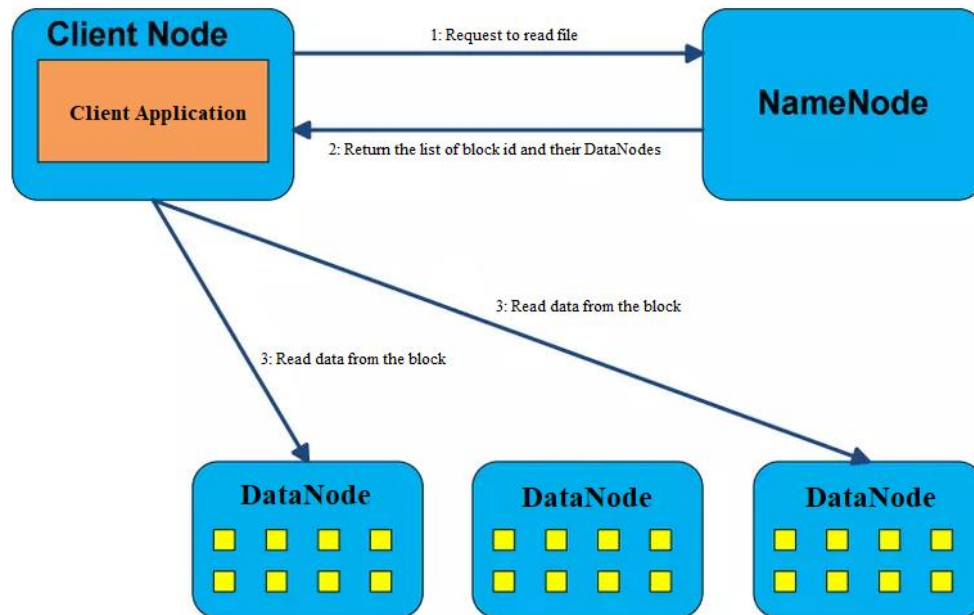


Figure 2: The read file process in HDFS

This section would demonstrate how a file can be read from the HDFS system. First of all, the client application would send a request to the NameNode. Afterward, NameNode will send back a list of blockID and their DataNode addresses before verifying the file requested is exist and is in a “Healthy State”.

After that, the client application would send the request to connect to the DataNode and execute a request to retrieve the blocks required to read, then close the connection to the DataNode. The client can read multiple replicas of a block inside DataNodes, but it would only read the block data from the nearest DataNode. All of these implementations will be hidden behind the API of Hadoop because Hadoop provides users with a set of APIs to interact with the HDFS without explicitly showing the process of connection between these nodes.

2.1.4.2 Create and write file on HDFS

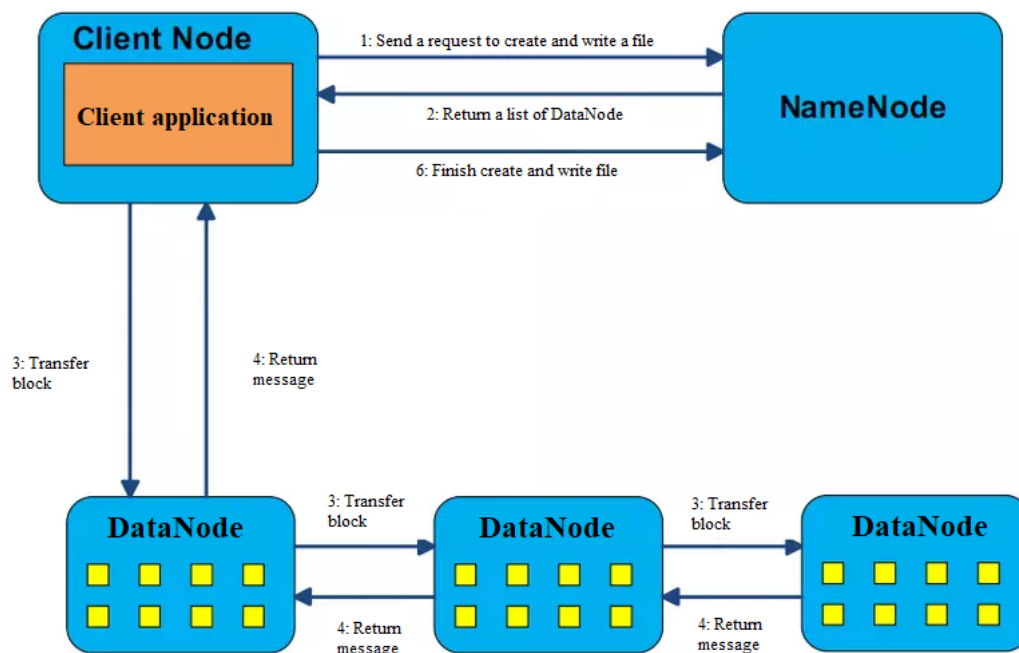


Figure 3: Write files process in HDFS

This section would demonstrate how a file is created and written in HDFS. At first, the client application will send a request to create and write a file to NameNode for it creates an empty file that has no block. After that, the NameNode would decide the list of DataNode containing the blocks and the replica of the blocks of that file back to the client application.

Afterward, the client application would divide the file into several blocks and encapsulate them into a package before transferring them to DataNode. The first DataNode receives the first package and will replicate the block before transferring it to the second DataNode to replicate the data. The second DataNode also does the same process as the first DataNode and transfers the block to the third DataNode. After the final DataNode replicates successfully, it will return a successful message to the second DataNode, and the second DataNode also sends back the same message to the first DataNode. Finally, the client will receive the successful message and send a message to NameNode to update and maintain the list of blocks of the file just created. The information for mapping block Id to DataNode will be automatically updated by the BlockReport receiving the DataNode. In the case one DataNode is inactive or a network error occurs, the client application will record all of the successful save of other DataNodes and retry the error DataNode.

2.2 MapReduce

The storage phase will be handled by the HDFS, while the process data phase will be handled by MapReduce. Apache Hadoop uses the MapReduce software architecture or programming model to create scalable, dependable, and fault-tolerant systems that process and analyze enormous data sets in parallel on large multi-node clusters of commodity hardware. There are two distinct phases used in data analysis and processing: the Map phase and the Reduce phase, each step done in parallel, each operating on sets of key-value pairs. The MapReduce model is quite similar to HDFS that it also exploits the master-slave architecture because MapReduce employs two kinds of daemons including JobTracker as master and TaskTracker as a slave node.

2.2.1 JobTracker

JobTracker's responsibilities include coordinating jobs and receiving jobs from the client program. It keeps track of MapReduce jobs released by TaskTracker on slave nodes. At first, JobTracker receives jobs from the client program and then asks the NameNode for allocating the required resources to process before assigning this job to Task Tracker on slave nodes. After receiving jobs from JobTracker, TaskTracker will respond with a heartbeat message to indicate the live status. If the JobTracker does not receive any heartbeat message in a specific time, the TaskTracker nodes which are assigned the task will be considered unfunctional and the job will be assigned to another TaskTracker by the JobTracker. For the Hadoop MapReduce service, JobTracker is a single point of failure; if it fails, all running jobs will be terminated.

2.2.2 TaskTracker

The TaskTracker will accept jobs from the assignment of JobTrackers and execute the MapReduce function. Based on a node's capability, each TaskTracker has a finite amount of "task slots." JobTracker can determine how many "task slots" are open in TaskTracker on a slave node thanks to the heartbeat protocol. According to how many open task slots there are, JobTracker's role is to assign the proper work to each TaskTracker. The heartbeat protocol also provides the JobTracker with the progress of the job in TaskTracker as well as TaskTracker state such as idle, in progress, or completed.

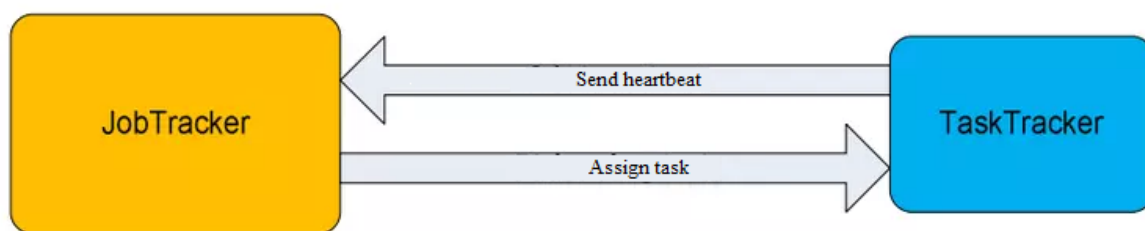


Figure 4: Interaction between JobTracker and TaskTracker

3.0 Method

The method used in this project is quantitative and qualitative research method. Two programs will be implemented using the data stored in HDFS. The data used for this project is collected through the website called "movielens.org" which stores the record of 10 million rating movies. The data is stored in a "rating.dat" file in unstructured form. There are 4 different columns separated by "::" notation including the "movie_id", "user_id", "rating", and "timestamp". The file will be uploaded to the HDFS system through the terminal commands.

There will be two programs that count the top 25 rated movies. The first program will be implemented using the MapReduce framework and the other will be implemented using the Spark framework to analyze and query the expected data. The first program will be implemented in Java and it would use Maven to convert from java files that are already compiled to .jar files and push them to the HDFS system through terminal command in order to use the MapReduce framework and library on Hadoop. The second program would be implemented in Python and it will utilize PySpark with Jupiter notebook in order to get data from HDFS as well as execute queries and write data in the HDFS system.

4.0 Implementation

4.1 Push input file to HDFS

The first requirement in order to implement any features of Hadoop is having input data. In this project, the data file will be retrieved from “movielens.org”. Afterward, developers can utilize some pre-defined commands in Hadoop to manipulate Hadoop distributed file system.

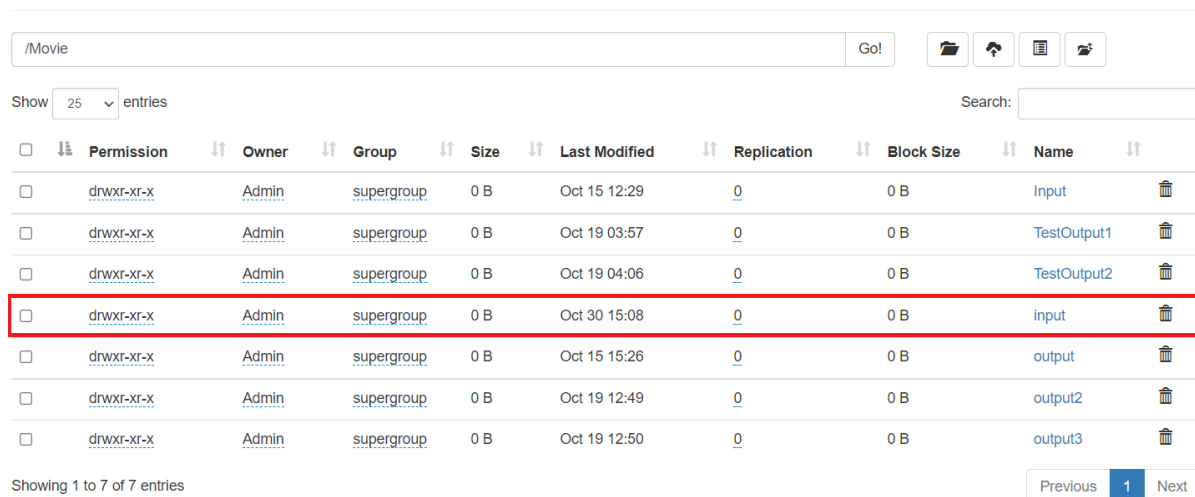
```
hadoop dfs -mkdir /Movie
hadoop dfs -mkdir /Movie/input
```

The first command is to create the folder project that will be used to store input files and output files. The second command is to create a new input directory in the Movie folder in HDFS.

```
C:\hadoop-3.3.0\sbin>hadoop dfs -mkdir /Movie/input
DEPRECATED: Use of this script to execute hdfs command is deprecated.
Instead use the hdfs command for it.
```

Figure 5: Successful creating a new folder on the terminal

Browse Directory



The screenshot shows the NameNode monitor web browser interface. At the top, there is a search bar with "/Movie" entered and a "Go!" button. Below the search bar, there are icons for file operations. The main area displays a table of directory entries. The table has columns for Permission, Owner, Group, Size, Last Modified, Replication, Block Size, and Name. The entry for "input" is highlighted with a red box. Below the table, there is a pagination bar showing "Showing 1 to 7 of 7 entries" and "Previous 1 Next".

Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
drwxr-xr-x	Admin	supergroup	0 B	Oct 15 12:29	0	0 B	Input
drwxr-xr-x	Admin	supergroup	0 B	Oct 19 03:57	0	0 B	TestOutput1
drwxr-xr-x	Admin	supergroup	0 B	Oct 19 04:06	0	0 B	TestOutput2
drwxr-xr-x	Admin	supergroup	0 B	Oct 30 15:08	0	0 B	input
drwxr-xr-x	Admin	supergroup	0 B	Oct 15 15:26	0	0 B	output
drwxr-xr-x	Admin	supergroup	0 B	Oct 19 12:49	0	0 B	output2
drwxr-xr-x	Admin	supergroup	0 B	Oct 19 12:50	0	0 B	output3

Figure 6: Successful creating a new folder in on the NameNode monitor web browser

Afterward, what needs to be fulfilled is to identify the location of the file that is required to push to HDFS with the folder directory in HDFS by typing the following command to the terminal:

```
hadoop fs -put <file_location> <HDFS_directory>
```

```
C:\hadoop-3.3.0\sbin>hadoop fs -put D:\swinburne\sem_2_y1\final\mapreduce\movie\input_data\ratings.dat \Movie\input
C:\hadoop-3.3.0\sbin>
```

Figure 7: Successful uploading of a file to the HDFS on the terminal

Browse Directory

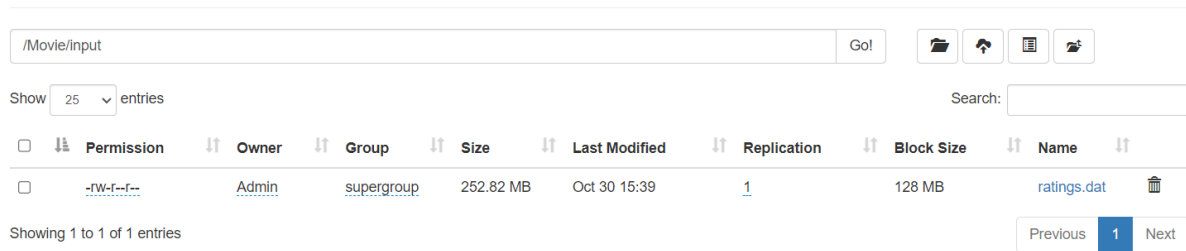


Figure 8: Successful uploading a file to the HDFS on the NameNode monitor web browser

Then, the content in the file can be verified through the terminal command using the command below:

```
hadoop dfs -cat \Movie\input\ratings.dat
```

4.2 Map phase

The first phase in the MapReduce algorithm is the mapping data phase. The input data first will be divided into smaller parts, then each part will be assigned to a mapper to process. The mapper would create other key-value pairs, referred to as intermediate key-value pairs using the input data set as a key-value combination according to the formulas below:

$$\langle k1, v1 \rangle \rightarrow \text{Map}() \rightarrow \text{list}(\langle k2, v2 \rangle)$$

In the counting rated movie among the 10 million rating program in Java, the first job of MapReduce is to map input into <key, value> pair. The form of original input will be in form of a string:

```
1::122::5::838985046
1::185::5::838983525
1::231::5::838983392
1::292::5::838983421
1::316::5::838983392
1::329::5::838983392
1::355::5::838984474
1::356::5::838983653
1::362::5::838984885
```

Figure 9: The first 9 lines of ratings.dat

The first column will be the “user_id”, the second column is labelled as “movie_id”, the third column is the “rating” and the final column will be the “timestamp” in form of seconds. In this input, only the second column will be used to count that identify the movie that is rated by a user. It will be mapped into a function to produce a <key, pair> value:

```

public static class CountMapper extends Mapper<Object, Text, Text, IntWritable> {
    private Text word = new Text();
    @Override
    // below is the function to get data and map it to key pair value
    public void map(Object key, Text value, Context context)
        throws IOException, InterruptedException {
        word.set(value.toString().split("::")[1]);
        // to take each parameter from a line of data
        context.write(word, ONE); //write to the file
    }
}

```

The map function will first divide the data in each line with the “::” sign and take the value of the second column only to be the key labeled as “word”, then map it into <word,1> used to count movies that are rated.

4.3 Shuffle and sort phase

Between the map and the reduce phase, there is a sub-phase called the shuffle and sort phase. This phase will be executed simultaneously with the map function, and it would sort and shuffle the words and their count at the same time. The sort functionality is grouping the list that has the same key, while the function of shuffle is to transfer the data from the mapper to reducers. This function will be executed automatically by Apache Hadoop so developers do not need to implement it.

4.4 Reduce phase

The Reduce job is to aggregate all inputs from the shuffle and apply them to reduce function and produce the final key-value pair, which is the expected output. In the final output, the counting rated movies program will produce a list of keys, and values, where the key is the “movie_id”, while the value will be the number that users rate for the movie according to the formula below:

$$< k2, \text{list}(v2) > \rightarrow \text{Reduce}() \rightarrow \text{list}(< k3, v3 >)$$

```

public static class CountReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
    @Override
    public void reduce(Text key, Iterable<IntWritable> values, Context context)
        throws IOException, InterruptedException {
        int counter = StreamSupport.stream(values.spliterator(), true)
            .mapToInt(IntWritable::get)
            .sum();
        // get total of all value groupby key
        context.write(key, new IntWritable(counter));
    }
}

```

From the code in the program, a counter will be used to calculate the total sum of the list(v2) in <k2, list (v2)> and transfer it to the “IntWritable” type. This type is utilized because Hadoop does not accept the normal int type in Java. Here is the output:

```

1 26449
10 16918
100 2640
1000 128
1001 23
1002 103
1003 1027
1004 977
1005 1128
1006 991

```

Figure 10: First 10 lines of the output

4.5 Finding the top 25 most-rated movies

There is no function to sort the list of “movie_id” and their “rating_count” in MapReduce. Hence the map and reduce function again need to implement to select the top 25 most-rated movies.

```

private static int k = 25;
public static class RankMapper extends Mapper<Text, Text, LongWritable, Text> {

    @Override
    public void map(Text item, Text vote, Context context)
        throws IOException, InterruptedException {
        context.write(new LongWritable(Long.parseLong(vote.toString())), item);
    }
}

```

One of the underlying functionalities of MapReduce is to arrange the output according to the value of the key (not the value). As a result, this can be utilized to sort the top 25 rated movies. Firstly, the vote in text form will be converted to “LongWritable” because Hadoop does not accept long type in Java, and then swap it with the key.

```

public static class RankReducer extends Reducer<LongWritable, Text, Text, LongWritable> {
    private int counter = 0;
    @Override
    public void reduce(LongWritable vote, Iterable<Text> items, Context context)
        throws IOException, InterruptedException {
        if (counter < k) { // display the ties at last item
            for (Text item : items) {
                context.write(item, vote); // flip the key and value
                counter++;
            }
        }
    }
}

```

The reduce function will aggregate all the <key, value> with the same key, and convert back to the original form which is <movie, rating_count>. Here is the final output with the left column as the “movie_id”, and the second column being rating counts:

296	34864
356	34457
593	33668
480	32631
318	31126
110	29154
457	28951
589	28948
260	28566
150	27035

Figure 11: First 10 lines of the output

4.6 Config assignment job in MapReduce

```
@Override
public int run(String[] args) throws Exception {
    Configuration conf = this.getConf();
    Job job = Job.getInstance(conf, "Counter");
    job.setJarByClass(RatingCounter.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    job.setInputFormatClass(TextInputFormat.class);
    job.setMapperClass(CountMapper.class);
    job.setCombinerClass(CountReducer.class);
    job.setReducerClass(CountReducer.class);
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    return job.waitForCompletion(true) ? 0 : 1;
}
```

The configuration in the "run" function will assign the job to the JobTracker and TaskTracker, as well as locate the input data by asking the NameNode. In addition, it assigns TaskTracker with map, shuffle, and reduce functions. Finally, an output directory will be announced to the NameNode and create an empty file, then write the data in that file. Here is the final output with the left column as the "movie_id", and the second column as "rating_counts":

4.7 Apache Spark

As an alternative to the conventional batch map and reduce methodology, Spark is built to run on top of Hadoop and may be used for real-time stream data processing and quick queries that complete their job in a matter of seconds. For executing apps, Spark offers two choices. The Scala language package includes an interpreter that enables users to utilize the Spark engine to conduct queries on huge data sets. Second, applications may be created as driver programs in Scala and delivered to the cluster's master node once they have been compiled. In this example, PySpark will be utilized to manipulate Spark SQL query and connect to HDFS on Hadoop.

4.8 PySpark program

The first implementation is to import all the required a package which is PySpark, start a PySpark session, and form the connection with the Hadoop server.

```
import pyspark;
from pyspark.sql import SparkSession;
from pyspark.sql.types import StructType
from pyspark.sql.functions import split
from pyspark.sql.functions import col, desc, concat, lit
from pyspark.sql import *

spark = SparkSession.builder.master("local").appName("Movie").getOrCreate()
```

Afterward, Spark will read the input data in HDFS and transform it into DataFrame. By doing this, the data can be converted to structured data by splitting each row of data with “::”:

```
booksdata = spark.read.text("hdfs://localhost:9000/Movie/Input/ratings.dat")
booksdata_1 = booksdata.withColumn('user_id', split(booksdata['value'], "::").getItem(0))\
    .withColumn('movie_id', split(booksdata['value'], "::").getItem(1))\
    .withColumn('rating', split(booksdata['value'], "::").getItem(2))\
    .withColumn('timestamp', split(booksdata['value'], "::").getItem(3))
booksdata_1 = booksdata_1.drop("value")
booksdata_1.show(truncate=False)
```

user_id	movie_id	rating	timestamp
1	122	5	838985046
1	185	5	838983525
1	231	5	838983392
1	292	5	838983421
1	316	5	838983392
1	329	5	838983392
1	355	5	838984474
1	356	5	838983653
1	362	5	838984885
1	364	5	838983707
1	370	5	838984596
1	377	5	838983834
1	420	5	838983834
1	466	5	838984679
1	480	5	838983653
1	520	5	838984679
1	539	5	838984068
1	586	5	838984068
1	588	5	838983339
1	589	5	838983778

only showing top 20 rows

Figure 12: First 20 lines of the table after converting data in Spark

Spark can handle big data files and processes the data 5 times faster than the MapReduce model. In addition, with Spark, we can perform SQL queries easily with a simple syntax.

```
booksdata_2 = booksdata_1.groupBy("movie_id")\
    .count()\
    .sort(col("movie_id"))
rating_count = booksdata_2.select(concat(col("movie_id")\
    .cast("string"), lit(' '), col("count"))\
    .cast("string"))\
    .alias("value")
rating_count.show()
```

```

+-----+
|   value|
+-----+
|  1 26449|
| 10 16918|
| 100 2640|
| 1000 128|
| 1001 23|
| 1002 103|
| 1003 1027|
| 1004 977|
| 1005 1128|
| 1006 991|
| 1007 1103|
| 1008 491|
| 1009 1598|
| 101 2226|
| 1010 1540|
| 1011 973|
| 1012 2039|
| 1013 2365|
| 1014 1000|
| 1015 1870|
+-----+

```

Figure 13: First 20 lines of the table after querying data in Spark

One of the problems when storing files using Spark in text form is that it only accepts one column string. That means the two columns need to be concatenated into one by using the “concat” function and then save to HDFS on Hadoop.

```
rating_count.write.save("hdfs://localhost:9000/Movie/output2/", format = 'text', mode = 'append')
```

With this, we have an output file that contains a list of “movie_id” with their rating counts.

4.9 Finding the top 25 most-rated movies by PySpark

In contrast to the MapReduce function, PySpark allows developers to sort the data using SQL and alias the column name:

```

booksdata_3 = booksdata_1.groupBy("movie_id" )\
    .count()\
    .sort(col("count" )\
        .desc())\
    .limit(25)

top_k_rate = booksdata_3.select(concat(col("movie_id")\
    .cast("string"),lit(' '),col("count"))\
    .cast("string"))\
    .alias("value"))

top_k_rate.show();

```



```

+-----+
| value|
+-----+
| 296 34864|
| 356 34457|
| 593 33668|
| 480 32631|
| 318 31126|
| 110 29154|
| 457 28951|
| 589 28948|
| 260 28566|
| 150 27035|
| 592 26996|
| 1 26449|
| 780 26042|
| 590 25912|
| 527 25777|
| 380 25381|
|1210 25098|
| 32 24397|
| 50 24037|
| 608 23794|
+-----+
only showing top 20 rows

```

Figure 14: First 20 lines of the final output in Spark

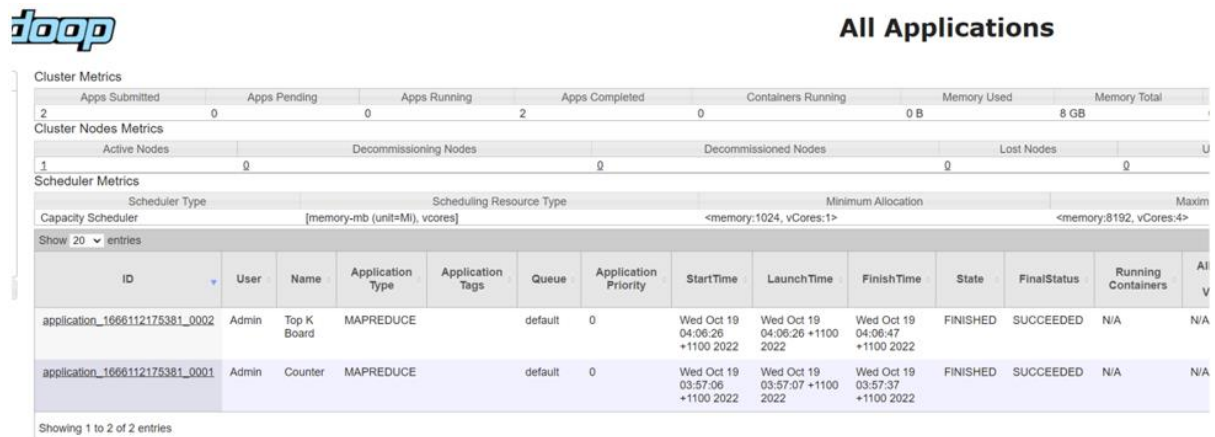
And then store it in HDFS on Hadoop but in a different file directory:

```
top_k_rate.write.save("hdfs://localhost:9000/Movie/output3/", format = 'text', mode = 'append')
```

5.0 Result

5.1 Result of implementation of Spark and MapReduce

The java program will be compiled and converted into a .jar file before being pushed to the HDFS. Afterward, the Hadoop system will run the program and convert the program into a job having a specific time to execute. The job will require a period to transform from the “ACCEPTED” state to the “FINISHED” state.



The screenshot shows the Hadoop Resource Manager Monitor Web Browser interface. The top section displays 'All Applications' with a table of cluster metrics. Below this, there are sections for 'Cluster Nodes Metrics' and 'Scheduler Metrics'. The main part of the interface is a table showing the details of two completed MapReduce jobs.

Cluster Metrics													
Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total							
2	0	0	2	0	0 B	8 GB							
Cluster Nodes Metrics													
Active Nodes	Decommissioning Nodes	Decommissioned Nodes	Lost Nodes										
1	0	0	0										
Scheduler Metrics													
Scheduler Type	Scheduling Resource Type	Minimum Allocation	Maximum Allocation										
Capacity Scheduler	[memory-mb (unit=Mi), vcores]	<memory:1024, vCores:1>	<memory:8192, vCores:4>										
Show 20 entries													
ID	User	Name	Application Type	Application Tags	Queue	Application Priority	StartTime	LaunchTime	FinishTime	State	FinalStatus	Running Containers	Allocated V
application_1666112175381_0002	Admin	Top K Board	MAPREDUCE		default	0	Wed Oct 19 04:06:26 +1100 2022	Wed Oct 19 04:06:26 +1100 2022	Wed Oct 19 04:06:47 +1100 2022	FINISHED	SUCCEEDED	N/A	N/A
application_1666112175381_0001	Admin	Counter	MAPREDUCE		default	0	Wed Oct 19 03:57:06 +1100 2022	Wed Oct 19 03:57:07 +1100 2022	Wed Oct 19 03:57:37 +1100 2022	FINISHED	SUCCEEDED	N/A	N/A
Showing 1 to 2 of 2 entries													





Figure 15: The succeeded jobs on the resource manager monitor web browser

As can be observed above, there are two jobs handled by a single node running in Hadoop in the “FINISHED” state which means it accomplished the program and created a new output file in the output folder. The total time to execute these jobs is 51 seconds.

Browse Directory

/Movie/output

Go!















Show

25

 entries

Search:

<input type="checkbox"/>	 Permission	 Owner	 Group	 Size	 Last Modified	 Replication	 Block Size	 Name	
<input type="checkbox"/>	-rw-r--r--	Admin	supergroup	0 B	Oct 15 14:13	1	128 MB	_SUCCESS	
<input type="checkbox"/>	-rw-r--r--	Admin	supergroup	91.21 KB	Oct 15 14:13	1	128 MB	part-r-00000	
<input type="checkbox"/>	drwxr-xr-x	Admin	supergroup	0 B	Oct 19 02:07	0	0 B	topk	

Showing 1 to 3 of 3 entries

Previous

1

Next

Figure 16: The succeeded output files created by MapReduce on the NameNode monitor web browser

The Spark framework also creates jobs for the Hadoop system and has a web interface to manipulate and monitor jobs.

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
5	save at NativeMethodAccessorImpl.java:0 save at NativeMethodAccessorImpl.java:0	2022/10/30 18:14:03	0.1 s	1/1 (1 skipped)	1/1 (12 skipped)
4	save at NativeMethodAccessorImpl.java:0 save at NativeMethodAccessorImpl.java:0	2022/10/30 18:13:58	4 s	1/1	12/12
3	save at NativeMethodAccessorImpl.java:0 save at NativeMethodAccessorImpl.java:0	2022/10/30 18:13:58	0.4 s	1/1 (2 skipped)	1/1 (13 skipped)
2	save at NativeMethodAccessorImpl.java:0 save at NativeMethodAccessorImpl.java:0	2022/10/30 18:13:57	0.3 s	1/1 (1 skipped)	1/1 (12 skipped)
1	save at NativeMethodAccessorImpl.java:0 save at NativeMethodAccessorImpl.java:0	2022/10/30 18:13:57	0.3 s	1/1 (1 skipped)	1/1 (12 skipped)
0	save at NativeMethodAccessorImpl.java:0 save at NativeMethodAccessorImpl.java:0	2022/10/30 18:13:51	6 s	1/1	12/12

Figure 17: The executed jobs run on PySpark Shell monitor web browser

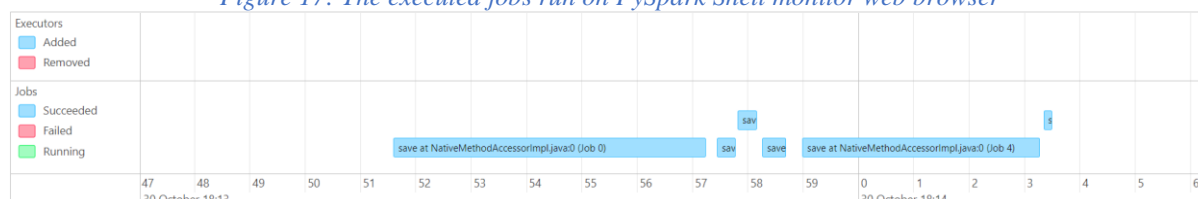






Figure 18: The event timeline for executed jobs run on PySpark Shell monitor web browser

As can be observed from the result of jobs execution above, it took around 11.1 seconds to execute 5 different jobs in PySpark which are 5 times faster compared to the MapReduce framework.

Browse Directory



/Movie/output3

Go!



Show 25 entries

Search:

<input type="checkbox"/>	Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name	<input type="checkbox"/>
<input type="checkbox"/>	-rw-r--r--	Admin	supergroup	0 B	Oct 30 18:14	3	128 MB	_SUCCESS	
<input type="checkbox"/>	-rw-r--r--	Admin	supergroup	248 B	Oct 30 18:14	3	128 MB	part-00000-b60a421a-5a3d-40f9-bc6f-ec5d135f6387-c000.txt	

Showing 1 to 2 of 2 entries

Previous

1

Next

Figure 19: The succeeded output files created by Spark on the NameNode monitor web browser

296	34864
356	34457
593	33668
480	32631
318	31126
110	29154
457	28951
589	28948
260	28566
150	27035
592	26996
1	26449
780	26042
590	25912
527	25777
380	253815
1210	25098
32	24397
50	24037
608	23794
377	23748
588	23531
2571	23229
1196	23091
47	22521

Figure 20: The final output after execution both of MapReduce and PySpark.

5.2 Spark vs MapReduce

The program that is written in both PySpark in Python and MapReduce in Java transforms the data to the expected output and generates jobs for the JobTracker to divide and assign them to the TaskTracker as well as access the data from the DataNode via the NameNode. As a result, there is a specific time to process the two same jobs. In this experiment, 5 tests have been conducted to compare the execution time of Spark and MapReduce and the visualization for time execution is below:

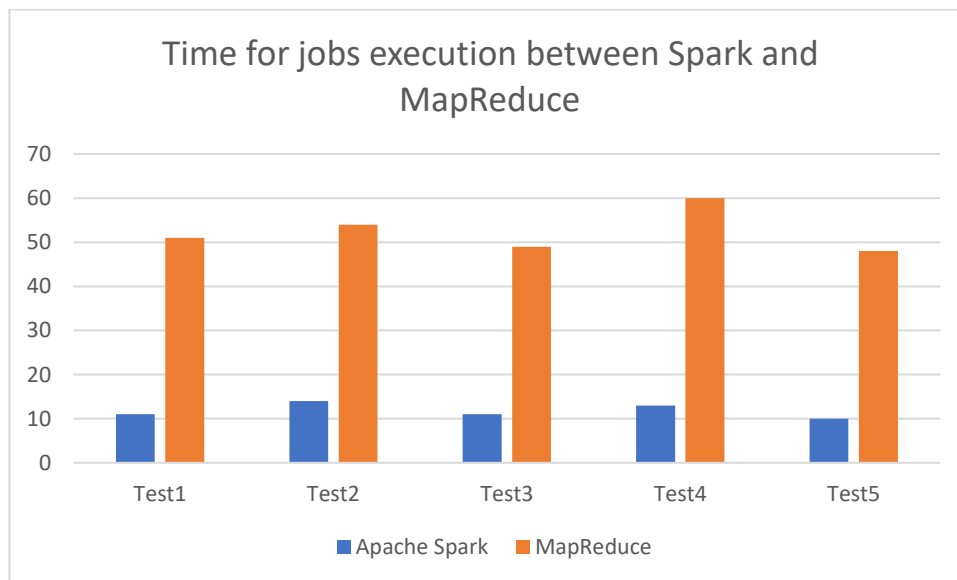


Table 1: The comparison between the time used to execute jobs between Spark and MapReduce

As can be observed from the illustration, it is clear that 5 tests demonstrate that Spark outperforms MapReduce in terms of processing data which is an average of 4.4 times faster compared to MapReduce.

5.3 Quantitative result

In terms of query simplicity, Spark would be also better than MapReduce because MapReduce has a complicated syntax, as well as developers, need to assign jobs and tell the program explicitly where to get the data from. In PySpark, it is clear that the number of codes would decrease significantly with simple syntax, and with the support of interactive queries, developers can check the validity and the correctness of the query immediately. While to validate the code in the program in MapReduce, developers need to convert it to a “.jar” file using Maven, then submit it to the HDFS and download the result to check the correctness of the query. Therefore, it can conclude that MapReduce is not suitable for applications that repeatedly reuse the same dataset which leads to inflexibility. However, Spark maintains active collections in memory for effective reusing which becomes an advantage compared to MapReduce.

6.0 Discussion

In this project, two programs have been implemented to compare the performance of the same task in different frameworks of Hadoop which are Spark and MapReduce. Apparently, from the result of experiments conducted in this project, it is undeniable to conclude the outperformance of Spark over MapReduce. From the perspective of developers, the research found that Spark has a friendlier syntax using Python and it is easier to develop with the assistance of Jupyter Notebook over MapReduce with Maven and Java. Verma et al. (2016) conducts a study comparing Spark and MapReduce in different aspects of the framework based on several metrics. They found that Hadoop MapReduce storage is the disk, while Spark relies heavily on cluster memory (RAM), which becomes one of the main reasons why Spark is outperform MapReduce in terms of processing and querying data. Another finding is that MapReduce is a batch-processing framework, which will process a couple of jobs periodically without the interference of developers. Spark also has this feature, with the addition of supporting interactive queries, which allow the user to examine the data when writing queries. One significant finding in their report includes that Spark is 100 times faster than MapReduce in the scope of processing big data. This can be seen in these experiments conducted mentioned above; however, due to the small scope of this project as well as the dataset is not big enough to demonstrate the 100 times faster of outperform of Spark over MapReduce.

7.0 Conclusion

With the support from the Apache Hadoop framework, all of these processes are hidden under the hood which allows developers to easily understand how to interact with MapReduce and HDFS. This paper discusses most of the processes behind the scene of the Hadoop distributed file system and MapReduce model. Additionally, the paper also mentions how they collaborate to handle big data sets. Although some tools such as Spark currently outperform MapReduce in terms of performance and availability, it is still important to understand the HDFS and MapReduce systems to implement the big data model and continue learning about other big data tools like Hadoop Hive, Apache Spark, and Hadoop Pig. In brief, these tools such as MapReduce and Spark can be still developed or become the fundamental infrastructures to construct innovative tools and models for the Hadoop ecosystem.

Word counts: 4028

Reference

- Borthakur, D. (2008).
HDFS architecture guide. *Hadoop apache project*, 53(1-13), 2.
https://docs.huihoo.com/apache/hadoop/1.0.4/hdfs_design.pdf
- Dittrich, J., & Quiané-Ruiz, J. A. (2012).
Efficient big data processing in Hadoop MapReduce. *Proceedings of the VLDB Endowment*, 5(12), 2014-2015. doi: 10.14778/2367502.2367562
- Ghazi, M. R., & Gangodkar, D. (2015).
Hadoop, MapReduce and HDFS: a developer's perspective. *Procedia Computer Science*, 48, 45-50. doi: 10.1016/j.procs.2015.04.108
- Karun, A. K., & Chitharanjan, K. (2013, April).
A review on hadoop—HDFS infrastructure extensions. In *2013 IEEE conference on information & communication technologies* (pp. 132-137).
doi: 10.1109/CICT.2013.6558077
- Maitrey, S., & Jha, C. K. (2015).
MapReduce: simplified data analysis of big data. *Procedia Computer Science*, 57, 563-571. doi: 10.1016/j.procs.2015.07.392
- Patel, A. B., Birla, M., & Nair, U. (2012, December).
Addressing big data problem using Hadoop and Map Reduce. In *2012 Nirma University International Conference on Engineering (NUICONE)* (pp. 1-5). IEEE. doi: 10.1109/NUICONE.2012.6493198
- Shafer, J., Rixner, S., & Cox, A. L. (2010, March).
The hadoop distributed filesystem: Balancing portability and performance. In *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)* (pp. 122-133). doi: 10.1109/ISPASS.2010.5452045
- Shvachko, K., Kuang, H., Radia, S., & Chansler, R. (2010, May).
The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)* (pp. 1-10). doi: 10.1109/MSST.2010.5496972.
- Verma, A., Mansuri, A. H., & Jain, N. (2016, March).
Big data management processing with Hadoop MapReduce and spark technology: A comparison. In *2016 symposium on colossal data analysis and networking (CDAN)* (pp. 1-4). doi: 10.1109/CDAN.2016.7570891.
- Zhai, Y., Tchaye-Kondi, J., Lin, K. J., Zhu, L., Tao, W., Du, X., & Guizani, M. (2021).
Hadoop perfect file: A fast and memory-efficient metadata access archive file to face small files problem in hdfs. *Journal of Parallel and Distributed Computing*, 156, 119-130. doi: 10.1016/j.jpdc.2021.05.011

Appendix

All the code for this project will be stored in the following link:

<https://drive.google.com/drive/folders/1st-IUUglPSHqSXMQUIU23fGj8n422HgxT?usp=sharing>