Tribhuvan University

Institute of Engineering

Thapathali Campus, Thapathali

Big Data Technologies Assignment 3

Submitted By:

Name: Raman Bhattarai

Roll No.: THA077BEI033

Submitted To:

Department of Electronics and Computer Engineering

Date: Feb 22, 2025

Hadoop is an open-source framework designed to store and process massive amounts of data efficiently. It provides a distributed computing environment, allowing for scalable and fault-tolerant data processing across clusters of computers. Developed originally by Apache Software Foundation, Hadoop has become a cornerstone technology in the big data ecosystem.

System Specifications:

The updated hardware and software specification of Virtual Machine system in VirtualBox is given below.

Hardware:

- CPU: 4 cores
- RAM: 5 GB
- Storage: 25 GB

Software:

- Operating System: Ubuntu 24.04.1
- Virtualization Platform: Oracle Virtual Machine (VM)

The initial hardware specifications of VM system were increased as the initial specifications were not sufficient to run Hadoop mapreduce.

Problem 1: Count the frequency of words in a given plain text file using hadoop framework.

Objectives:

- To count the frequency of words in a text file using hadoop.

Problem Definition:

The goal of this lab is to count how often each word appears in a plain text file using the Hadoop framework. Handling large text files can be difficult with traditional methods, so Hadoop helps by breaking the task into smaller parts and processing them in parallel. The process involves reading the text file, splitting it into individual words, and counting how many times each word appears. Hadoop's MapReduce model will be used to distribute the workload across multiple machines, making the word count process faster and more efficient.

Step-by-step approach to solve the Problem:

1. Start hadoop server

```
Raman@Ubuntu:~$ hadoop version
Hadoop 3.4.0
Source code repository git@github.c
```

```
Raman@Ubuntu:~$ start-dfs.sh
Starting namenodes on [localhost]
Starting datanodes
Starting secondary namenodes [Ubuntu]
Raman@Ubuntu:~$ start-yarn.sh
Starting resourcemanager
Starting nodemanagers
```

2. Check for running services

```
Raman@Ubuntu:~$ jps
7091 ResourceManager
7223 NodeManager
7595 Jps
6875 SecondaryNameNode
6635 DataNode
6493 NameNode
```

3. Temporarily set hadoop class path

```
Raman@Ubuntu:~$ export HADOOP_CLASSPATH=$(hadoop classpath)
Raman@Ubuntu:~$ echo $HADOOP_CLASSPATH
/home/Raman/hadoop-3.4.0//etc/hadoop:/home/Raman/hadoop-3.4.0//sha
adoop-3.4.0//share/hadoop/common/*:/home/Raman/hadoop-3.4.0//share
```

4. Create directory on HDFS system

```
Raman@Ubuntu:~$ hadoop fs -mkdir /WordCountExample
```

5. Create a directory inside it for the input

```
Raman@Ubuntu:~$ hadoop fs -mkdir /WordCountExample/Input
```

6. Make directory on your local system inside Desktop

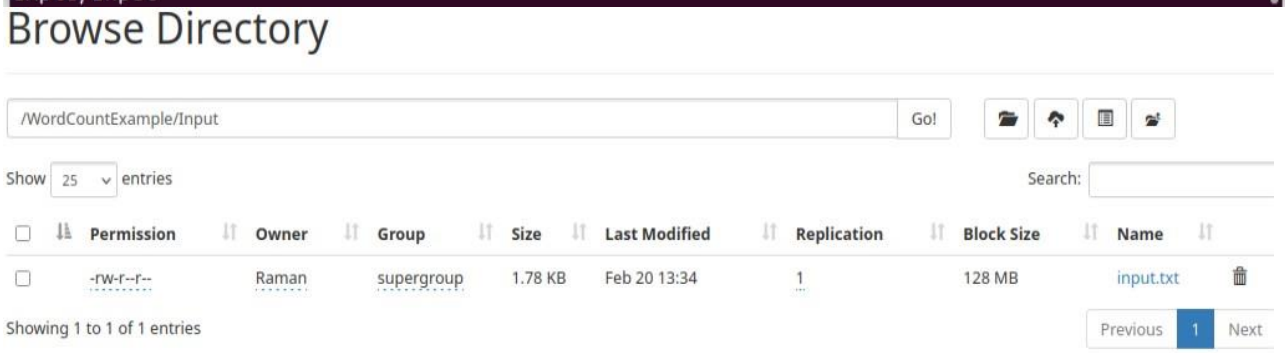
```
Raman@Ubuntu:~$ sudo mkdir /home/Raman/WordCountExample/input_data
```

7. Give all permision to the base directory recusively

```
Raman@Ubuntu:~$ sudo chmod 777 -R /home/Raman/WordCountExample
Raman@Ubuntu:~$ sudo chmod 777 -R /home/Raman/WordCountExample/input_data
```

8. Upload the given input file from your local system to hadoop system

```
Raman@Ubuntu:~$ hadoop fs -put '/home/Raman/WordCountExample/input_data/input.txt' /WordCountEx
ample/Input
```



9. Change the current directory to the Example directory and move wordcount example code
   a. Develop a Java MapReduce program based on the Apache Hadoop WordCount example.
   b. The Mapper function handles the text line by line, breaking it into words and emitting each word with a count of 1.
   c. The Reducer function combines these counts to calculate the total frequency of each word.

Source Code:

```
import java.io.IOException;
import java.util.StringTokenizer;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
```

```
public class WordCount {

    public static class TokenizerMapper extends Mapper<Object, Text,
Text, IntWritable> {
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();

        public void map(Object key, Text value, Context context) throws
IOException, InterruptedException {
            StringTokenizer itr = new
StringTokenizer(value.toString());
            while (itr.hasMoreTokens()) {
                word.set(itr.nextToken());
                context.write(word, one);
            }
        }
    }

    public static class IntSumReducer extends Reducer<Text,
IntWritable, Text, IntWritable> {
        private IntWritable result = new IntWritable();

        public void reduce(Text key, Iterable<IntWritable> values,
Context context) throws IOException, InterruptedException {
            int sum = 0;
            for (IntWritable val : values) {
                sum += val.get();
            }
            result.set(sum);
            context.write(key, result);
        }
    }

    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "word_count_example");

        job.setJarByClass(WordCount.class);
        job.setMapperClass(TokenizerMapper.class);
        job.setCombinerClass(IntSumReducer.class);
        job.setReducerClass(IntSumReducer.class);

        job.setMapOutputValueClass(IntWritable.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        job.setInputFormatClass(TextInputFormat.class);
        job.setOutputFormatClass(TextOutputFormat.class);
```

```
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

10. Compile the Java Code

```
Raman@Ubuntu:~/WordCountExample$ javac -classpath ${HADOOP_CLASSPATH} -d '/home/Raman/WordCount
Example/classfile' '/home/Raman/WordCountExample/WordCount.java'
```

11. Put the output files in one jar file

```
Raman@Ubuntu:~/WordCountExample$ jar -cvf wcexample.jar -C classfile/ .
added manifest
adding: WordCount$IntSumReducer.class(in = 1755) (out= 749)(deflated 57%)
adding: WordCount.class(in = 1812) (out= 928)(deflated 48%)
adding: WordCount.java(in = 3551) (out= 1055)(deflated 70%)
adding: WordCount$TokenizerMapper.class(in = 1752) (out= 763)(deflated 56%)
```

12. Run the jar file on Hadoop

```
Raman@Ubuntu:~/WordCountExample$ hadoop jar /home/Raman/WordCountExample/wcexample.jar
 WordCount /WordCountExample/Input /WordCountExample/Output
```

13. See the Output

```
Raman@Ubuntu:~/WordCountExample$ hadoop dfs -cat /WordCountExample/Output/*
```

Input Data (A text file):

The term “big data” refers to data that is so large, fast or complex that it’s difficult or impossible to process using traditional methods. The act of accessing and storing large amounts of information for analytics has been around a long time. But the concept of big data gained momentum in the early 2000s when industry analyst Doug Laney articulated the now-mainstream definition of big data as the three V’s:

Volume: Organizations collect data from a variety of sources, including business transactions, smart (IoT) devices, industrial equipment, videos, social media and more. In the past, storing it would have been a problem – but cheaper storage on platforms like data lakes and Hadoop have eased the burden.

Velocity: With the growth in the Internet of Things, data streams in to businesses at an unprecedented speed and must be handled in a timely manner. RFID tags, sensors and smart meters are driving the need to deal with these torrents of data in near-real time.

Variety: Data comes in all types of formats – from structured, numeric data in traditional databases to unstructured text documents, emails, videos, audios, stock ticker data and financial transactions.

In addition to the increasing velocities and varieties of data, data flows are unpredictable – changing often and varying greatly. It’s challenging, but businesses need to know when something is trending in social media, and how to manage daily, seasonal and event-triggered peak data loads.

Veracity refers to the quality of data. Because data comes from so many different sources, it’s difficult to link, match, cleanse and transform data across systems. Businesses need to connect and correlate relationships, hierarchies and multiple data linkages. Otherwise, their data can quickly spiral out of control.

Output:

Word	Count	Word	Count	Word	Count
(IoT)	1	databases	1	of	11
2000s	1	data”	1	often	1
Because	1	deal	1	on	1
Businesses	1	definition	1	or	2
But	1	devices,	1	out	1
Data	1	different	1	past,	1
Doug	1	difficult	2	peak	1
Hadoop	1	documents,	1	platforms	1
In	2	driving	1	problem	1
Internet	1	early	1	process	1
It’s	1	eased	1	quality	1
Laney	1	emails,	1	quickly	1
Organizations	1	equipment,	1	refers	2
Otherwise,	1	event-triggered	1	relationships,	1

RFID	1	fast	1	seasonal	1
The	2	financial	1	sensors	1
Things,	1	flows	1	smart	2
Variety:	1	for	1	so	2
Velocity:	1	formats	1	social	2
Veracity	1	from	3	something	1
Volume:	1	gained	1	sources,	2
V’s:	1	greatly.	1	speed	1
With	1	growth	1	spiral	1
a	4	handled	1	stock	1
accessing	1	has	1	storage	1
across	1	have	2	storing	2
act	1	hierarchies	1	streams	1
addition	1	how	1	structured,	1
all	1	impossible	1	systems.	1
amounts	1	in	8	tags,	1
an	1	including	1	term	1
analyst	1	increasing	1	text	1
analytics	1	industrial	1	that	2
and	13	industry	1	the	11
are	2	information	1	their	1
around	1	is	2	these	1
articulated	1	it	1	three	1
as	1	it’s	2	ticker	1
at	1	know	1	time.	2
audios,	1	lakes	1	timely	1
be	1	large	1	to	11
been	2	large,	1	torrents	1
big	2	like	1	traditional	2
burden.	1	link,	1	transactions,	1
business	1	linkages.	1	transactions.	1
businesses	2	loads.	1	transform	1
but	2	long	1	trending	1
can	1	manage	1	types	1
challenging,	1	manner.	1	unprecedented	1
changing	1	many	1	unpredictable	1
cheaper	1	match,	1	unstructured	1
cleanse	1	media	1	using	1
collect	1	media,	1	varieties	1
comes	2	meters	1	variety	1
complex	1	methods.	1	varying	1
concept	1	momentum	1	velocities	1
connect	1	more.	1	videos,	2
control.	1	multiple	1	when	2
correlate	1	must	1	with	1
daily,	1	near-real	1	would	1
data	15	need	3	–	3
data,	1	now-mainstream	1	“big	1
data.	1	numeric	1		

Problem 2: Find the average/mean age of male and female died in titanic disaster.

Objectives:

- To find mean age of male and female that died in titanic disaster from given titanic dataset.

Problem Definition:

The goal of this lab is to calculate the average (mean) age of male and female passengers who died in the Titanic disaster. The dataset contains information about passengers, including their age, gender, and survival status. To solve this problem, we need to filter the data to include only those who did not survive, then separate them by gender and compute the average age for each group.

Step-by-step approach to solve the Problem:

1. Start hadoop server

```
Raman@Ubuntu:~$ hadoop version
Hadoop 3.4.0
Source code repository git@github.c
```

```
Raman@Ubuntu:~$ start-dfs.sh
Starting namenodes on [localhost]
Starting datanodes
Starting secondary namenodes [Ubuntu]
Raman@Ubuntu:~$ start-yarn.sh
Starting resourcemanager
Starting nodemanagers
```

2. Check for running services

```
Raman@Ubuntu:~$ jps
7091 ResourceManager
7223 NodeManager
7595 Jps
6875 SecondaryNameNode
6635 DataNode
6493 NameNode
```

3. Temporarily set hadoop class path

```
Raman@Ubuntu:~$ export HADOOP_CLASSPATH=$(hadoop classpath)
Raman@Ubuntu:~$ echo $HADOOP_CLASSPATH
/home/Raman/hadoop-3.4.0//etc/hadoop:/home/Raman/hadoop-3.4.0//sha
adoop-3.4.0//share/hadoop/common/*:/home/Raman/hadoop-3.4.0//share
```

4. Create directory on HDFS system

```
Raman@Ubuntu:~$ hadoop fs -mkdir /TitanicProblem
```

5. Create a directory inside it for the input

```
Raman@Ubuntu:~$ hadoop fs -mkdir /TitanicProblem/Input
```

6. Make directory on your local system inside Desktop

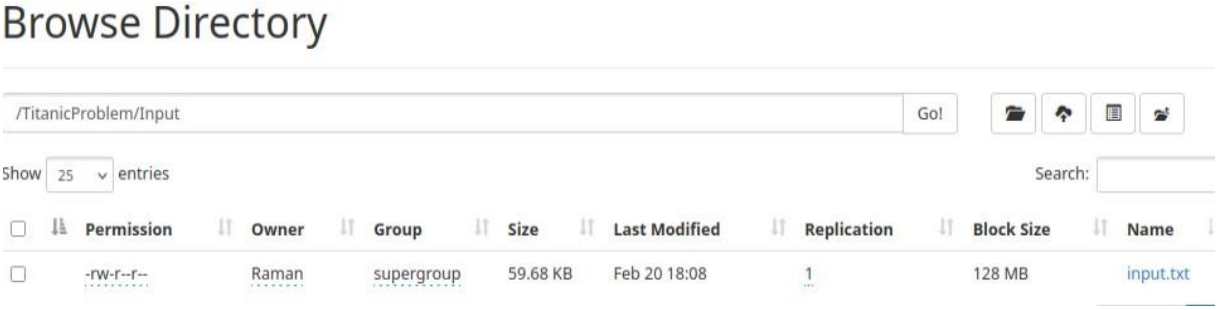
```
Raman@Ubuntu:~$ sudo mkdir /home/Raman/TitanicProblem/
Raman@Ubuntu:~$ sudo mkdir /home/Raman/TitanicProblem/input_data/
```

7. Give all permision to the base directory recusively

```
Raman@Ubuntu:~$ sudo chmod 777 -R /home/Raman/TitanicProblem
Raman@Ubuntu:~$ sudo chmod 777 -R /home/Raman/TitanicProblem/input_data
```

8. Upload the given input file from your local system to hadoop system

```
Raman@Ubuntu:~$ hadoop fs -put '/home/Raman/TitanicProblem/input_data/input.txt' /TitanicProblem/Input
```



9. Change the current directory to the Example directory

```
Raman@Ubuntu:~$ cd /home/Raman/TitanicProblem
```

10. Move titanic example code
    a. Develop a Java-based MapReduce program to compute the average age of deceased Titanic passengers, categorized by gender. The program processes the dataset, excludes survivors, and focuses on calculating the mean age for deceased males and females.
    b. The Mapper processes each input line, filters for deceased passengers (Survived == 0), and emits a key-value pair where the key is the gender (Sex) and the value is the passenger's age.
    c. The Reducer groups the data by gender, calculates the total age and the number of deceased passengers for each gender, and computes the average age by dividing the total age by the count.

Source Code:

```
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.FloatWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;

public class TitanicMeanAge {
```

```
    // Mapper Class
    public static class MeanAgeMapper extends Mapper<Object, Text,
Text, FloatWritable> {
        private Text gender = new Text();
        private FloatWritable age = new FloatWritable();

        public void map(Object key, Text value, Context context) throws
IOException, InterruptedException {
            String line = value.toString();

            // Split by comma (no space after comma in typical CSV)
            String[] str = line.split(",");

            // Check for valid number of columns
            if (str.length > 6) {
                String survived = str[1].trim(); // Survived (0 or 1)
                String sex = str[4].trim();      // Gender
                String ageStr = str[5].trim();   // Age

                // Check if the person died (0) and age is a valid
number
                if (survived.equals("0") && !ageStr.isEmpty()) {
                    try {
                        float ageValue = Float.parseFloat(ageStr);  //
Convert age to float
                        gender.set(sex);
                        age.set(ageValue);
                        context.write(gender, age);
                    } catch (NumberFormatException e) {
                        // Ignore rows where age is not a valid number
                    }
                }
            }
        }
    }

    // Reducer Class
    public static class MeanAgeReducer extends Reducer<Text,
FloatWritable, Text, FloatWritable> {
        private FloatWritable result = new FloatWritable();

        public void reduce(Text key, Iterable<FloatWritable> values,
Context context) throws IOException, InterruptedException {
            float sum = 0;
            int count = 0;

            // Calculate sum and count of ages
            for (FloatWritable val : values) {
                sum += val.get();
```

```
                count++;
            }

            // Avoid division by zero
            if (count > 0) {
                result.set(sum / count);
                context.write(key, result);
            }
        }
    }

    // Main Method
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "titanic_mean_age");

        job.setJarByClass(TitanicMeanAge.class);
        job.setMapperClass(MeanAgeMapper.class);
        job.setReducerClass(MeanAgeReducer.class);

        // Set Mapper output key-value types
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(FloatWritable.class);

        // Set Reducer output key-value types
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(FloatWritable.class);

        job.setInputFormatClass(TextInputFormat.class);
        job.setOutputFormatClass(TextOutputFormat.class);

        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

11. Compile the Java Code

```
Raman@Ubuntu:~/TitanicProblem$ javac -classpath ${HADOOP_CLASSPATH} -d '/home/Raman/TitanicProblem/classfile'
'/home/Raman/TitanicProblem/TitanicMeanAge.java'
```

12. Put the output files in one jar file

```
Raman@Ubuntu:~/TitanicProblem$ jar -cvf titanicmeanage.jar -C classfile/
```

13. Run the jar file on Hadoop

```
Raman@Ubuntu:~/TitanicProblem$ hadoop jar '/home/Raman/TitanicProblem/titanicmeanage.jar' TitanicMeanAge
/TitanicProblem/Input /TitanicProblem/Output
```

14. See the Output

```
Raman@Ubuntu:~/TitanicProblem$ hadoop dfs -cat /TitanicProblem/Output/*
```

Output:

```
Raman@Ubuntu:~/TitanicProblem$ hadoop dfs -cat /TitanicProblem/Output/*
WARNING: Use of this script to execute dfs is deprecated.
WARNING: Attempting to execute replacement "hdfs dfs" instead.

female  25.046875
male    31.618055
```

Conclusion:

In conclusion, both tasks demonstrated Hadoop's powerful capabilities for large-scale data processing. While implementing the word frequency and Titanic age calculation programs, the provided code contained errors such as improper tokenization in the word count program and issues with age parsing and division by zero in the Titanic program. Despite these challenges, the tasks emphasized Hadoop's flexibility in handling complex data and the importance of debugging in large-scale data processing applications.