

Experiment No.4	
Experiment on Hadoop Map-Reduce	
Date of Performance:	
Date of Submission:	



<u>AIM</u>: -To write a program to implement a word count program using MapReduce.

THEORY:

WordCount is a simple program which counts the number of occurrences of each word in a given text input data set. WordCount fits very well with the MapReduce programming model making it a great example to understand the Hadoop Map/Reduce programming style. The implementation consists of three main parts:

- 1. Mapper
- 2. Reducer
- 3. Driver

Step-1. Write a Mapper

A Mapper overrides the —mapl function from the Class "org.apache.hadoop.mapreduce.Mapper" which provides <key, value> pairs as the input. A Mapper implementation may output <key,value> pairs using the provided Context.

Input value of the WordCount Map task will be a line of text from the input data file and the key would be the line number line_number, line_of_text>. Map task outputs <word, one> for each word in the line of text. Pseudo-code void Map (key, value) { for each word x in value:

```
output.collect(x,1);
}
```

Step-2. Write a Reducer

A Reducer collects the intermediate <key,value> output from multiple map tasks and assemble a single result. Here, the WordCount program will sum up the occurrence of each word to pairs as <word, occurrence>. Pseudo-code

```
void Reduce (keyword, <list of value>){ for each
x in <list of value>:
sum+=x;
final_output.collect(keyword, sum);
}
```

CSL702: Big Data Analytics Lab



Code:

```
import java.io.IOException; import
java.util.StringTokenizer; import
org.apache.hadoop.io.IntWritable; import
org.apache.hadoop.io.LongWritable; import
org.apache.hadoop.io.Text; import
org.apache.hadoop.mapreduce.Mapper; import
org.apache.hadoop.mapreduce.Reducer; import
org.apache.hadoop.conf.Configuration; import
org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.fs.Path; public class WordCount
{
public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> { public
void map(LongWritable key, Text value, Context context) throws
IOException, Interrupted Exception {
String line = value.toString();
StringTokenizer tokenizer = new StringTokenizer(line);
while
              (tokenizer.hasMoreTokens())
value.set(tokenizer.nextToken()); context.write(value,
new IntWritable(1));
CSL702: Big Data Analytics Lab
```

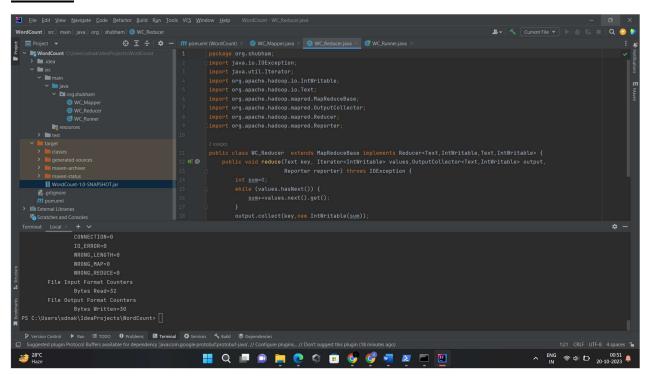


```
} }
public static class Reduce extends Reducer<Text,IntWritable,Text,IntWritable> {
public void reduce(Text key, Iterable<IntWritable> values,Context context)
throws IOException, InterruptedException { int sum=0; for(IntWritable x: values)
\{ sum+=x.get(); \}
context.write(key, new IntWritable(sum));
} }
public static void main(String[] args) throws Exception {
Configuration conf= new Configuration(); Job job = new
Job(conf,"My Word Count Program");
job.setJarByClass(WordCount.class);
job.setMapperClass(Map.class);
job.setReducerClass(Reduce.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);
job.setInputFormatClass(TextInputFormat.class);
job.setOutputFormatClass(TextOutputFormat.class);
Path outputPath = new Path(args[1]);
//Configuring the input/output path from the filesystem into the job
FileInputFormat.addInputPath(job, new Path(args[0])); FileOutputFormat.setOutputPath(job,
new Path(args[1]));
//deleting the output path automatically from hdfs so that we don't have to delete
it explicitly
outputPath.getFileSystem(conf).delete(outputPath); //exiting
the job only if the flag value becomes false
CSL702: Big Data Analytics Lab
```

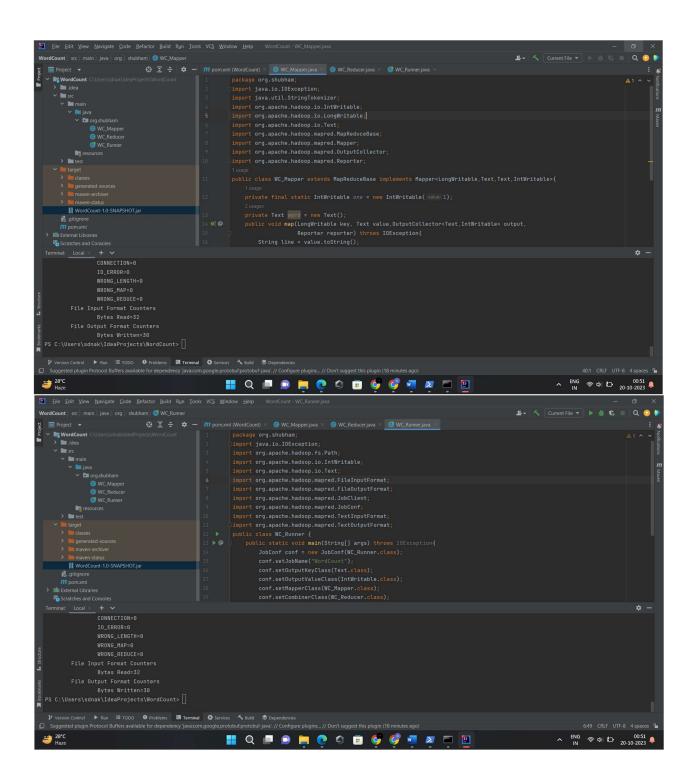


```
System.exit(job.waitForCompletion(true) ? 0 : 1);
}
```

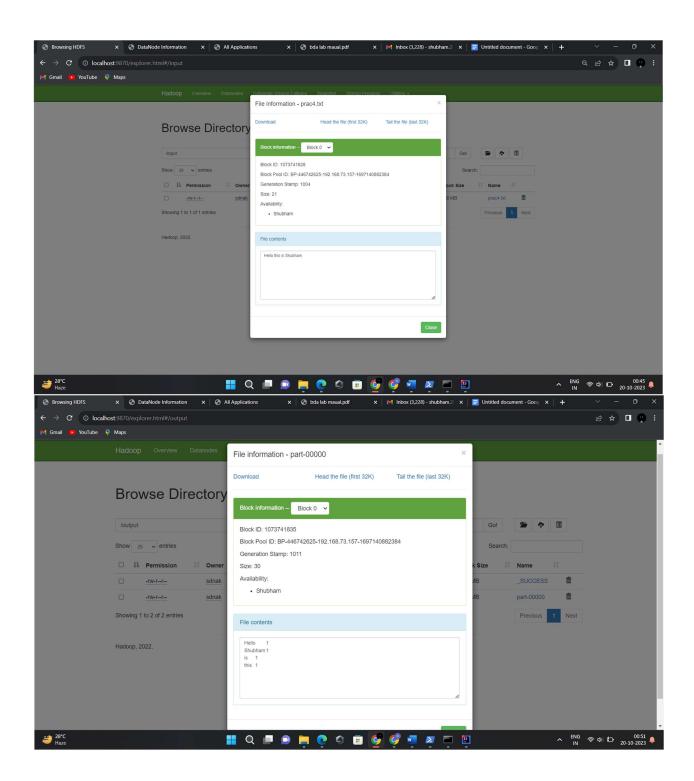
OUTPUT:













CONCLUSION:

The experiment aimed at implementing a word count program using MapReduce yielded successful results. MapReduce demonstrated its ability to scale and efficiently process extensive datasets by distributing tasks across multiple nodes for parallel execution. It also showcased its fault tolerance, ensuring the integrity of data processing in distributed systems. MapReduce's simplicity, characterized by its straightforward mapper and reducer functions, makes it accessible to a broad spectrum of developers. The practical relevance of this experiment extends to more intricate data processing tasks, such as log analysis and machine learning. By incorporating performance optimizations like combiners and partitioners, the program's efficiency can be further improved. In essence, this experiment established a robust foundation in distributed computing, a highly valuable skill in today's data-driven landscape.

CSL702: Big Data Analytics Lab