ASSIGNMENT 2

PROBLEM STATEMENT: -
Develop a MapReduce program to calculate the frequency of a given word in a given file.

OBJECTIVE:
1. Develop a MapReduce program to calculate the frequency of a given word.
2.Understand the MapReduce programming model and its key components (Mapper and Reducer).
3. Implement the logic for word frequency calculation in the Mapper and Reducer functions.

Hardware Requirements –
• A Hadoop cluster for distributed computing.
• Sufficient computational resources on the cluster to handle the input file.

Software Requirements –
• Hadoop Installation: Ensure that Hadoop is installed and configured on the cluster.
• Java Development Kit (JDK): MapReduce programs are typically written in Java, so JDK is required for development.
• Text Editor or Integrated Development Environment (IDE): Use a text editor or IDE to write and edit the MapReduce program.
• Input File: The file for which the word frequency is to be calculated.

THEORY:

• **Hadoop** is a software framework for easily writing applications which process vast amounts of data (multi-terabyte data-sets) in-parallel on large clusters (thousands of nodes) of commodity hardware in a reliable, fault-tolerant manner. It was popularized by Google and is widely used in open-source implementations like Apache Hadoop.

• **A MapReduce** job usually splits the input data-set into independent chunks which are processed by the map tasks in a completely parallel manner. The framework sorts the outputs of the maps, which are then input to the reduce tasks. Typically both the input and the output of the job are stored in a file-system. The framework takes care of scheduling tasks, monitoring them and re-executes the failed tasks.

• In the **Map phase**, the input data is divided into smaller chunks and processed independently in parallel across multiple nodes in a distributed computing environment. Each chunk is transformed or

"mapped" into key-value pairs by applying a user-defined function. The output of the Map phase is a set of intermediate key-value pairs.

• The **Reduce phase** follows the Map phase. It gathers the intermediate key-value pairs generated by the Map tasks, performs data shuffling to group together pairs with the same key, and then applies a user-defined reduction function to aggregate and process the data. The output of the Reduce phase is the final result of the computation.

• The MapReduce framework operates exclusively on pairs, that is, the framework views the input to the job as a set of pairs and produces a set of pairs as the output of the job, conceivably of different types.

• Input and Output types of a MapReduce job: (input) -> map -> -> combine -> -> reduce -> (output)

• **MapReduce** allows for efficient processing of large-scale datasets by leveraging parallelism and distributing the workload across a cluster of machines. It simplifies the development of distributed data processing applications by abstracting away the complexities of parallelization, data distribution, and fault tolerance, making it an essential tool for big data processing in the Hadoop ecosystem.

• **Word Count** is a simple application that counts the number of occurrences of each word in a given input set. This works with a local-standalone, pseudo-distributed or fully-distributed Hadoop installation.

• The text from the input text file is tokenized into words to form a key value pair with all the words present in the input text file. The key is the word from the input file and value is '1'.

• For instance if you consider the sentence "An elephant is an animal". The mapper phase in the Word Count example will split the string into individual tokens i.e. words. In this case, the entire sentence will be split into 5 tokens (one for each word) with a value 1 as shown below –

• Key-Value pairs from Hadoop Map Phase Execution-

(an,1)

(elephant,1)

(is,1)

(an,1)

(animal,1)

• Hadoop WordCount Example- Shuffle Phase Execution: After the map phase execution is completed successfully, shuffle phase is executed automatically wherein the key-value pairs generated in the map phase are taken as input and then sorted in alphabetical order. After the shuffle phase is executed from the WordCount example code, the output will look like this –

(an,1)

(an,1)

(animal,1)

(elephant,1)

(is,1)

•Hadoop WordCount Example- Reducer Phase Execution: In the reduce phase, all the keys are grouped together and the values for similar keys are added up to find the occurrences for a particular word. It is like an aggregation phase for the keys generated by the map phase. The reducer phase takes the output of shuffle phase as input and then reduces the key-value pairs to unique keys with values added up. In our example "An elephant is an animal." is the only word that appears twice in the sentence. After the execution of the reduce phase of MapReduce WordCount example program, appears as a key only once but with a count of 2 as shown below –

(an, 2)

(animal, 1)

(elephant, 1)

(is, 1)

• This is how the MapReduce word count program executes and outputs the number of occurrences of a word in any given input file. An important point to note during the execution of the WordCount example is that the mapper class in the WordCount program will execute completely on the entire input file and not just a single sentence. Suppose if the input file has 15 lines then the mapper class will split the words of all the 15 lines and form initial key value pairs for the entire dataset. The reducer execution will begin only after the mapper phase is executed successfully.

Code –

Mapper Code –

```
// Importing libraries import java.io.IOException;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.LongWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapred.MapReduceBase;

import org.apache.hadoop.mapred.Mapper;

import org.apache.hadoop.mapred.OutputCollector;

import org.apache.hadoop.mapred.Reporter;

public class WCMapper extends MapReduceBase implements Mapper<LongWritable, Text, Text, IntWritable> {

// Map function

public void map(LongWritable key, Text value, OutputCollector<Text

IntWritable> output, Reporter rep) throws IOException

{

String line = value.toString();

// Splitting the line on spaces

for (String word : line.split(" "))

{

if (word.length() > 0)

{

output.collect(new Text(word), new IntWritable(1));

} } } }
```

Reducer Code:

```
// Importing libraries import java.io.IOException;

import java.util.Iterator;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapred.MapReduceBase;

import org.apache.hadoop.mapred.OutputCollector;

import org.apache.hadoop.mapred.Reducer;

import org.apache.hadoop.mapred.Reporter;

public class WCReducer extends MapReduceBase implements Reducer<Text, IntWritable, Text, IntWritable> {

// Reduce function

public void reduce(Text key, Iterator value,

OutputCollector output, Reporter rep) throws IOException

{

int count = 0;

// Counting the frequency of each words

while (value.hasNext())

{

IntWritable i = value.next();

count += i.get();

 }

output.collect(key, new IntWritable(count));

}}
```

Driver Code:

```
// Importing libraries

import java.io.IOException;

import org.apache.hadoop.conf.Configured;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapred.FileInputFormat;

import org.apache.hadoop.mapred.FileOutputFormat;

import org.apache.hadoop.mapred.JobClient;

import org.apache.hadoop.mapred.JobConf;

import org.apache.hadoop.util.Tool;

import org.apache.hadoop.util.ToolRunner;

public class WCDriver extends Configured implements Tool

{

public int run(String args[]) throws IOException

{

if (args.length < 2)

{

System.out.println("Please give valid inputs");

return -1;

}

JobConf conf = new JobConf(WCDriver.class);

FileInputFormat.setInputPaths(conf,newPath(args[0]));
```

```
FileOutputFormat.setOutputPath(conf,newPath(args[1])); conf.setMapperClass(WCMapper.class);

conf.setReducerClass(WCReducer.class);

conf.setMapOutputKeyClass(Text.class);

conf.setMapOutputValueClass(IntWritable.class);

conf.setOutputKeyClass(Text.class);

conf.setOutputValueClass(IntWritable.class);

JobClient.runJob(conf);

return 0;

}


// Main Method

public static void main(String args[]) throws Exception

{

int exitCode = ToolRunner.run(new WCDriver(), args);

System.out.println(exitCode);

}

}
```

CONCLUSION:

In this way we have develop a MapReduce program to calculate the frequency of a given word in a given file.

ORAL QUESTION:

1. What are the key phases in a MapReduce program?
2. How do you handle intermediate key-value pairs between the Mapper and Reducer?
3. What challenges might arise when dealing with very large files in MapReduce?
4. Can you explain the concept of shuffling and sorting in MapReduce and its relevance here?

Q. Develop a MapReduce program to calculate the frequency of a given word in a given file.

Wordcount.py

```
import re
from multiprocessing import Pool

WORD_RE = re.compile(r"[\w']+")

def read_file(filename):
    with open(filename, 'r') as file:
        return file.readlines()

def mapper(line):
    word_count = {}
    for word in WORD_RE.findall(line):
        word_count[word.lower()] = word_count.get(word.lower(), 0) + 1
    return word_count

def reducer(mapped_counts):
    reduced_counts = {}
    for word_count in mapped_counts:
        for word, count in word_count.items():
            reduced_counts[word] = reduced_counts.get(word, 0) + count
    print(reduced_counts)
    return reduced_counts

def main(filename, target_word):
    lines = read_file(filename)
    with Pool() as pool:
        mapped_counts = pool.map(mapper, lines)
    reduced_counts = reducer(mapped_counts)

    # Get the frequency of the target word
    target_frequency = reduced_counts.get(target_word.lower(), 0)
```

```
    print(f"The frequency of '{target_word}' in the file is: {target_frequency}")


if __name__ == "__main__":
    filename = input("Enter the file name: ")
    target_word = input("Enter the word to find frequency: ")
    main(filename, target_word)
```

samplefile.txt

The quick brown fox jumps over the lazy dog. The lazy dog yawns and stretches. The fox looks back and smiles at the dog. Then, the fox continues its journey through the forest. The quick brown fox is a clever animal. It knows how to survive in the wild. The lazy dog, on the other hand, prefers to relax and enjoy life. Life is simple for the lazy dog. The quick brown fox and the lazy dog are good friends. They often play together in the meadow. Sometimes, they chase each other around the trees. Other times, they simply lie down and bask in the sun. But no matter what they do, they always have fun together.

Output:

```
PS D:\Learning only> & C:/ProgramData/Python310/python.exe "d:/Learning only/CL4/practical2.py"
Enter the file name: CL4\practical2.txt
Enter the word to find frequency: the
{'the': 17, 'quick': 3, 'brown': 3, 'fox': 5, 'jumps': 1, 'over': 1, 'lazy': 5, 'dog': 6, 'yawns': 1, 'and': 5, 'stretches': 1, 'looks': 1, 'back': 1, 'smile
s': 1, 'at': 1, 'then': 1, 'continues': 1, 'its': 1, 'journey': 1, 'through': 1, 'forest': 1, 'is': 2, 'a': 1, 'clever': 1, 'animal': 1, 'it': 1, 'knows': 1,
 'how': 1, 'to': 2, 'survive': 1, 'in': 3, 'wild': 1, 'on': 1, 'other': 3, 'hand': 1, 'prefers': 1, 'relax': 1, 'enjoy': 1, 'life': 2, 'simple': 1, 'for': 1,
 'are': 1, 'good': 1, 'friends': 1, 'they': 5, 'often': 1, 'play': 1, 'together': 2, 'meadow': 1, 'sometimes': 1, 'chase': 1, 'each': 1, 'around': 1, 'trees'
: 1, 'times': 1, 'simply': 1, 'lie': 1, 'down': 1, 'bask': 1, 'sun': 1, 'but': 1, 'no': 1, 'matter': 1, 'what': 1, 'do': 1, 'always': 1, 'have': 1, 'fun': 1}
The frequency of 'the' in the file is: 17
```