ASSIGNMENT 4

PROBLEM STATEMENT:

Develop a MapReduce program to find the grades of students.

OBJECTIVE:

1. Implement a MapReduce program to calculate grades for students based on their scores.
2. Understand the distributed processing nature of MapReduce in the context of grading.

Hardware Requirements

• A cluster of machines for distributed processing (number of nodes depends on the scale of data).

Software Requirements:

• Hadoop (or any other MapReduce framework).

Data set

• Student data with scores (e.g., CSV file with student ID, name, and scores in different subjects).

Libraries or modules used

• Hadoop MapReduce libraries.

Theory

• MapReduce Paradigm

MapReduce is a programming model and processing technique designed for large-scale data processing across distributed clusters. It divides a computational task into two main phases: the Map phase and the Reduce phase. In the context of grading students, MapReduce can efficiently handle the computation of average scores and assignment of grades across a large dataset.

• Mapper Function

The mapper function is responsible for processing each input record and emitting intermediate keyvalue pairs. In the case of student grading, the input data likely consists of records in the form of "student_name, subject, score." The mapper function extracts the relevant information, specifically the student name, subject, and score, and emits key-value pairs where the student name is the key and a tuple containing the subject and score is the value.

• Shuffling and Sorting

The framework automatically shuffles and sorts the intermediate key-value pairs, grouping them by key. In our student grading example, this means that all records related to a specific student are brought together, ready for processing in the next phase.

• **Reducer Function**

The reducer function takes the grouped key-value pairs and performs the necessary computations to determine the final output. In the context of grading students, the reducer calculates the average score for each student by iterating through the tuples of subjects and scores associated with that student. Once the average score is computed, the reducer assigns a grade based on predefined criteria (e.g., A for scores above 90, B for scores between 80 and 90, and so on). The final output of the reducer is a key-value pair where the key is the student name, and the value is the assigned grade.

• **Scalability and Parallel Processing**

One of the key strengths of MapReduce is its ability to scale horizontally, meaning it can efficiently process large datasets by distributing the computation across multiple nodes in a cluster. Each mapper and reducer operates independently on its subset of data, allowing for parallel processing and significantly reducing the time required for computation.

• **Flexibility and Extensibility**

The MapReduce model is flexible and extensible, making it suitable for a wide range of data processing tasks. In the student grading example, the same MapReduce framework can be adapted for different datasets and grading criteria by modifying the mapper and reducer functions.

Algorithm of work

• **Map phase**

Read student data, emit student ID as the key, and scores as values.

• **Reduce phase**

Calculate grades based on scores and emit the final result.

Applications

• Use cases include automated grading systems for educational institutions.

Code

1. Mapper:

```
Public class Mapper extends Mapper
{
Public void map(LongWritable key, Text value, Context context) throws IOException,
InterruptedException {
String line = value.toString();
String[] fields = line.split(","); // assuming comma-separated values
String studentId = fields[0];
int marks = Integer.parseInt(fields[1]); // assuming second field is marks
Text outputKey = new Text(studentId);
IntWritable outputValue = new IntWritable(marks);
context.write(outputKey, outputValue);
} }
```

This mapper reads each line of the input data containing student ID and marks separated by a comma. It then:

• Splits the line into fields based on the comma separator.

• Extracts the student ID and parses the marks as an integer.

• Creates a key-value pair where the key is the student ID and the value is the marks.

• Emits the key-value pair to the reducer.

2. Reducer:

Java

```
public class Reducer extends Reducer
{
public void reducer (Text key, Iterable values, Context context) throws IOException,
InterruptedException {
int totalMarks = 0;
for (IntWritable mark : values)
{
totalMarks += mark.get();
}
String grade = calculateGrade(totalMarks); // Replace with your logic for calculating grades
context.write(key, new Text(grade));
}
private String calculateGrade(int totalMarks)
{ // Replace with your specific grade calculation logic
if (totalMarks >= 90) {
return
"A";
} else
if (totalMarks >= 80) {
return
"B";
} else
if (totalMarks >= 70) {
return
"C"; } else
if (totalMarks >= 60) {
return
"D"; } else {
return "F";
} } }
```

This reducer receives key-value pairs from the mapper, where the key is the student ID and the values are individual marks. It:

• Iterates through the marks for each student and calculates the total marks.

• Uses your logic (replace calculateGrade(int totalMarks)) to determine the grade based on the total marks.

• Creates a new key-value pair where the key is the student ID and the value is the calculated grade.

3. Driver:

```
public class Driver
{
Public static void main(String[] args) throws Exception {
Configuration conf = new Configuration();
Job job = Job.getInstance(conf, "Student Grades");
job.setJarByClass(Driver.class);
job.setMapperClass(Mapper.class);
job.setReducerClass(Reducer.class);
job.setInputFormatClass(TextInputFormat.class); job.setOutputFormatClass(TextOutputFormat.class);
FileInputFormat.addInputPath(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job,newPath(args[1])); System.exit(job.waitForCompletion(true) ? 0 :
1);
}
```

}

This driver configures and submits the MapReduce job:

• Sets the job name and JAR file containing the mapper and reducer classes.

• Specifies the mapper and reducer classes to be used.

• Defines the input format (text file) and output format (text file).

• Sets the input and output paths for the job based on command-line arguments.

• Submits the job and waits for completion, exiting with an appropriate code.

4. **Running the program**:

• Save the code in separate files for Mapper, Reducer, and Driver.

• Compile the files using javac command.

• Run the program using the hadoop jar command, specifying the JAR file, input path, and output path as arguments.

• The output file will contain student IDs and their respective grades.

5. Adapting the program:

• Replace the comma-separated assumption with your actual data format.

• Modify the calculateGrade method to reflect your specific grading system.

• Adjust the input and output formats based on your needs.

CONCLUSION:

In this way we have develop a MapReduce program to find the grades of students.

ORAL QUESTION:

1. How do you handle scenarios where the grading criteria need to be adjusted or updated?
2. How do you ensure the correctness and efficiency of the MapReduce program?
3. What would be the output format of this MapReduce job?

Q. Develop a MapReduce program to find the grades of students

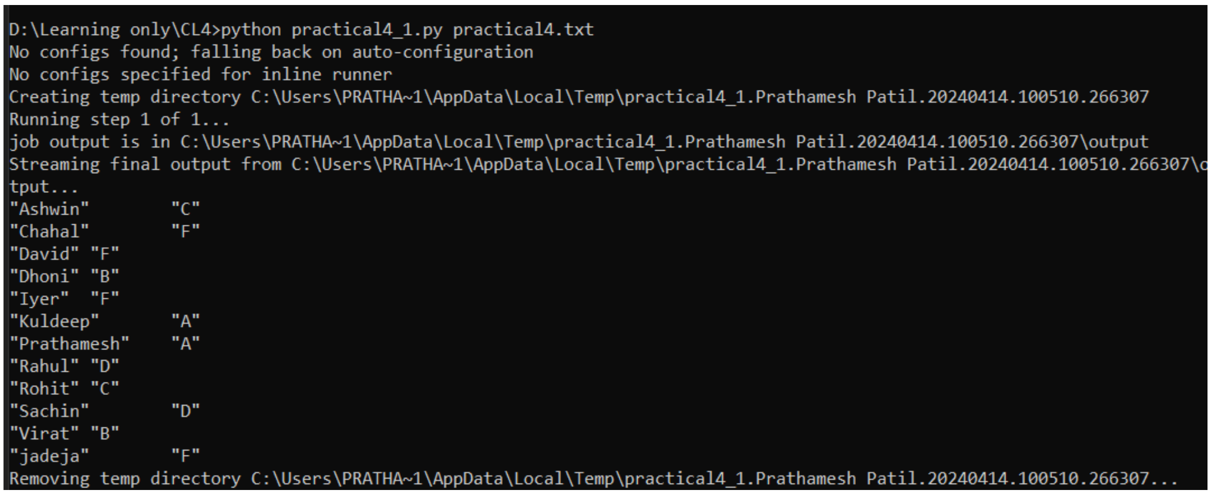
GradeCalculator.py

```
from mrjob.job import MRJob

class GradeCalculator(MRJob):
    def mapper(self, _, line):
        # Split the input line into name and score
        name, score = line.split(',')
        score = int(score)

        # Emit the name and score
        yield name, score

    def reducer(self, key, values):
        # Calculate the average score
        total_score = 0
        num_scores = 0
        for score in values:
            total_score += score
            num_scores += 1
        average_score = total_score / num_scores

        # Determine the grade based on the average score
        if average_score >= 90:
            grade = 'A'
        elif average_score >= 80:
            grade = 'B'
        elif average_score >= 70:
            grade = 'C'
        elif average_score >= 60:
            grade = 'D'
        else:
            grade = 'F'
        # Emit the name and grade

        yield key, grade


if __name__ == '__main__':

    GradeCalculator.run()
```

samplefile.txt

```
Prathamesh,95

Rohit,75

Virat,82

Sachin,68

Dhoni,88

Ashwin,73

Kuldeep,91

David,55

jadeja,50

Rahul,60

Iyer,45

Chahal,40
```

Output:



```
D:\Learning only\CL4>python practical4_1.py practical4.txt
No configs found; falling back on auto-configuration
No configs specified for inline runner
Creating temp directory C:\Users\PRATHA~1\AppData\Local\Temp\practical4_1.Prathamesh Patil.20240414.100510.266307
Running step 1 of 1...
job output is in C:\Users\PRATHA~1\AppData\Local\Temp\practical4_1.Prathamesh Patil.20240414.100510.266307\output
Streaming final output from C:\Users\PRATHA~1\AppData\Local\Temp\practical4_1.Prathamesh Patil.20240414.100510.266307\o
tput...
"Ashwin"        "C"
"Chahal"        "F"
"David" "F"
"Dhoni" "B"
"Iyer"  "F"
"Kuldeep"       "A"
"Prathamesh"    "A"
"Rahul" "D"
"Rohit" "C"
"Sachin"        "D"
"Virat" "B"
"jadeja"        "F"
Removing temp directory C:\Users\PRATHA~1\AppData\Local\Temp\practical4_1.Prathamesh Patil.20240414.100510.266307...
```