**UNIVERSITY SCHOOL OF**

**INFORMATION**

**COMMUNICATION AND**

**TECHNOLOGY**



**BIG DATA ANALYTICS LAB**

**SUBJECT CODE: ITE-403P**

**BACHELOR OF TECHNOLOGY**

**IN**

**COMPUTER SCIENCE AND ENGINEERING**

**SUBMITTED TO:                                    SUBMITTED BY:**

Dr. Anuradha Chug Pranav Jain

00516403221

B.Tech (CSE-7th Semester)

2021-25

**INDEX PAGE**

|  |  |  |  |
| --- | --- | --- | --- |
| **S. No** | **List of Experiments** | **Page No** | **Sign** |
| 1 | Install Apache Hadoop. | 3-6 |  |
| 2 | Develop a map reduce program to calculate the frequency of a given word in a given file. | 7-10 |  |
| 3 | Develop a map reduce program to find the maximum temperature in each year. | 11-12 |  |
| 4 | Develop a map reduce program to find the grade of students. | 13-16 |  |
| 5 | Develop a map reduce program to implement matrix multiplication. | 17-20 |  |
| 6 | Develop a map reduce program to find the maximum electrical consumption in each year given electrical consumption for each month in each year. | 21-23 |  |
| 7 | Develop a map reduce program to analyze the weather data set and print whether the day is shiny or cool. | 24-26 |  |
| 8 | Develop a map reduce program to find the tags associated with each movie by analyzing movie lens data. | 27-29 |  |
| 9 | Develop a map reduce program to analyze Uber data set to find the days on which each basement has more trips using the following data set. The uber data set consists of four columns they are: Dispatching, base, no. date active, vehicle trips. | 30-32 |  |
| 10 | Develop a map reduce program to analyze titanic dataset to find the average age of the people (both male and female) who died in the tragedy. How many people survived in each class. | 33-35 |  |

**PRACTICAL-1**

**Aim:** Install Apache Hadoop.

**Theory:**

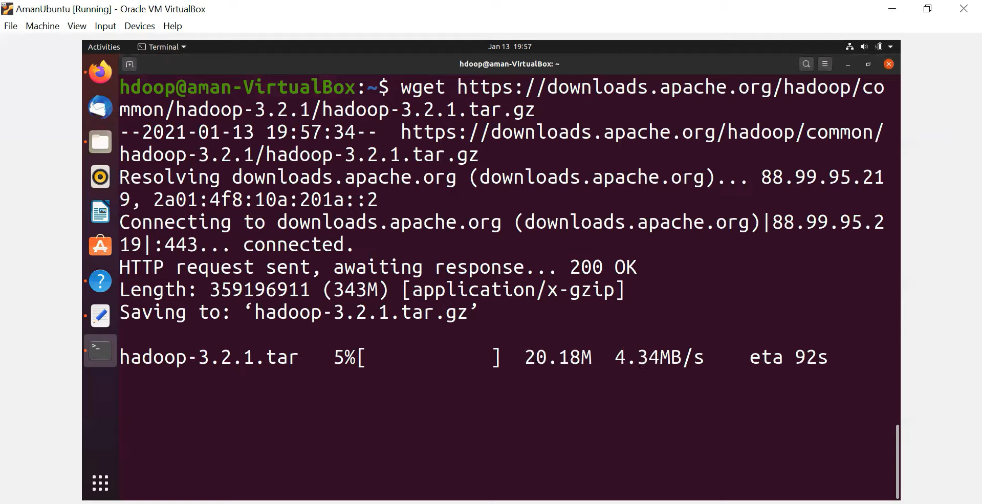
Apache Hadoop is a framework for distributed storage and processing of large datasets using a cluster of commodity hardware. Here's how it relates to Ubuntu:

1. Installation: You can install Apache Hadoop on Ubuntu by downloading the Hadoop binaries from the official Apache Hadoop website or by using package managers like apt. Ubuntu provides a convenient environment for setting up Hadoop clusters due to its ease of installation and robust package management system.
2. Compatibility: Hadoop is compatible with Ubuntu, meaning it can run smoothly on Ubuntu servers or desktops without major compatibility issues. This allows users to leverage the features of Ubuntu along with the capabilities of Hadoop for big data processing.
3. Resource Management: Ubuntu provides tools and utilities for managing system resources effectively, which can be beneficial when running Hadoop clusters. Resource management is crucial in Hadoop environments to ensure optimal performance and utilization of cluster resources.
4. Security: Ubuntu offers robust security features that can be leveraged to secure Hadoop clusters. This includes configuring firewalls, setting up user permissions, and enabling encryption to protect data stored and processed by Hadoop.
5. Maintenance: Ubuntu's regular updates and long term support (LTS) releases make it suitable for maintaining Hadoop clusters over extended periods. Updates can be easily applied to both Ubuntu and Hadoop components to ensure stability, security, and compatibility.
6. Community Support: Both Apache Hadoop and Ubuntu have vibrant communities that provide support, documentation, and resources for users. This makes it easier for users to troubleshoot issues, seek advice, and stay updated with the latest developments in both technologies.

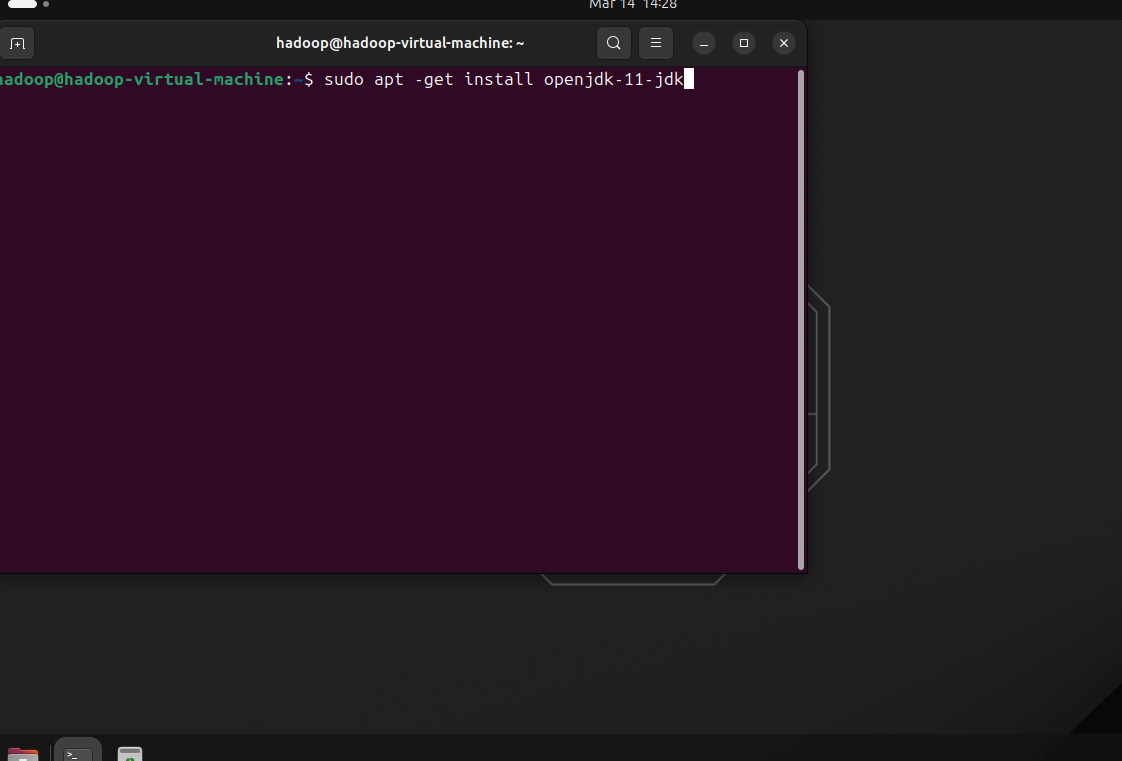
In summary, Ubuntu provides a reliable and flexible platform for deploying, managing, and maintaining Apache Hadoop clusters, making it a popular choice for organizations looking to harness the power of big data analytics.

**Steps:**

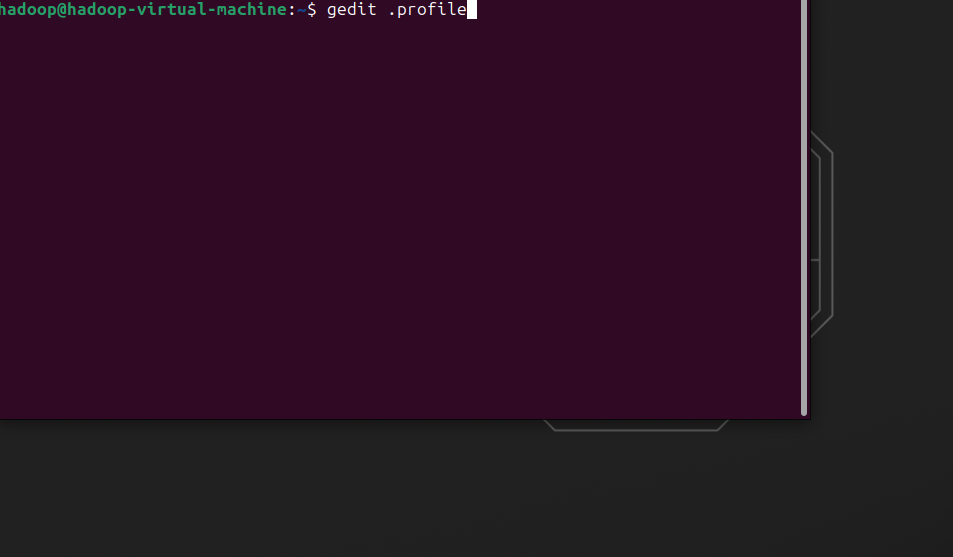
1. Install Hadoop in the virtual machine.

****

1. Download Java SE Development Kit

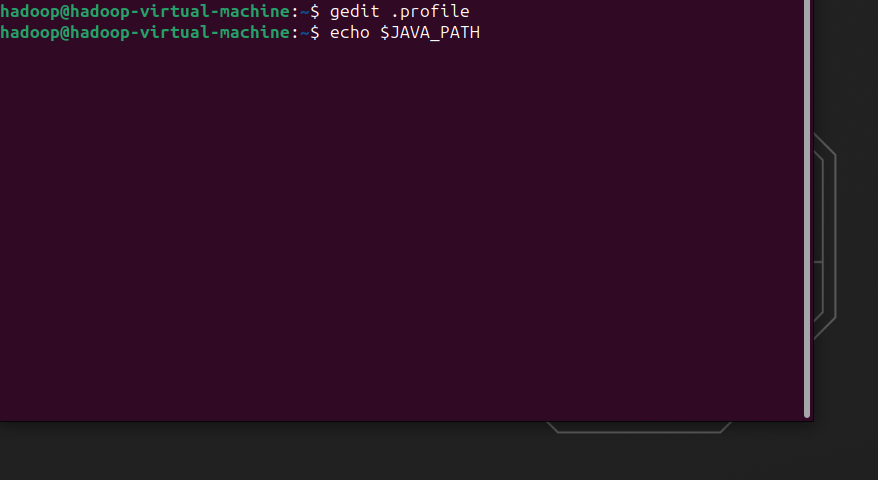


1. Set Environment Variable.

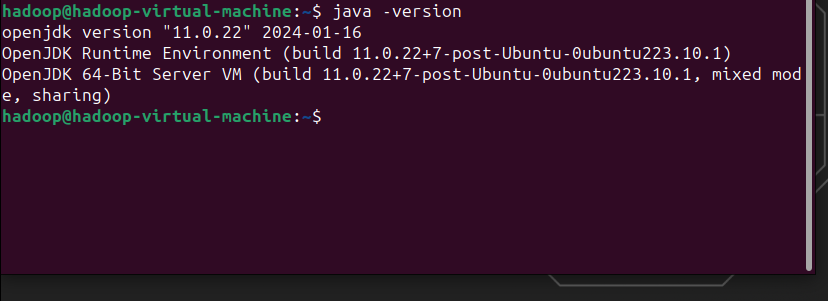




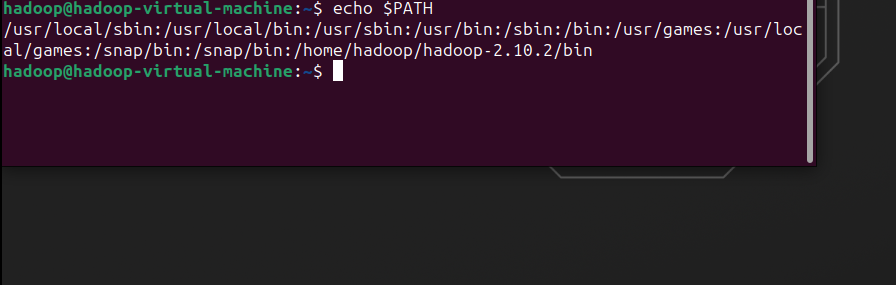
1. Check Java Path



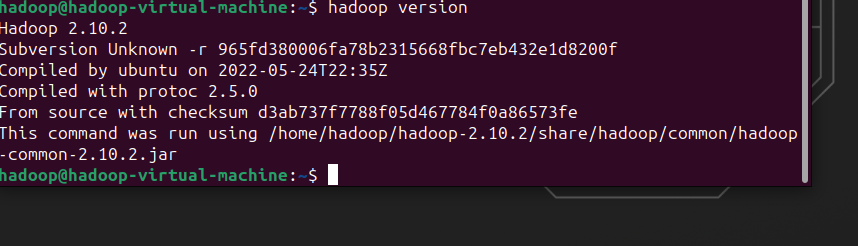
1. Check the JAVA version installed.



1. Check the Path for Hadoop.



1. Check Hadoop version.



**PRACTICAL 2**

**Aim:** Develop a mapreduce program to calculate the frequency of a given word in a given file.

**Theory:**

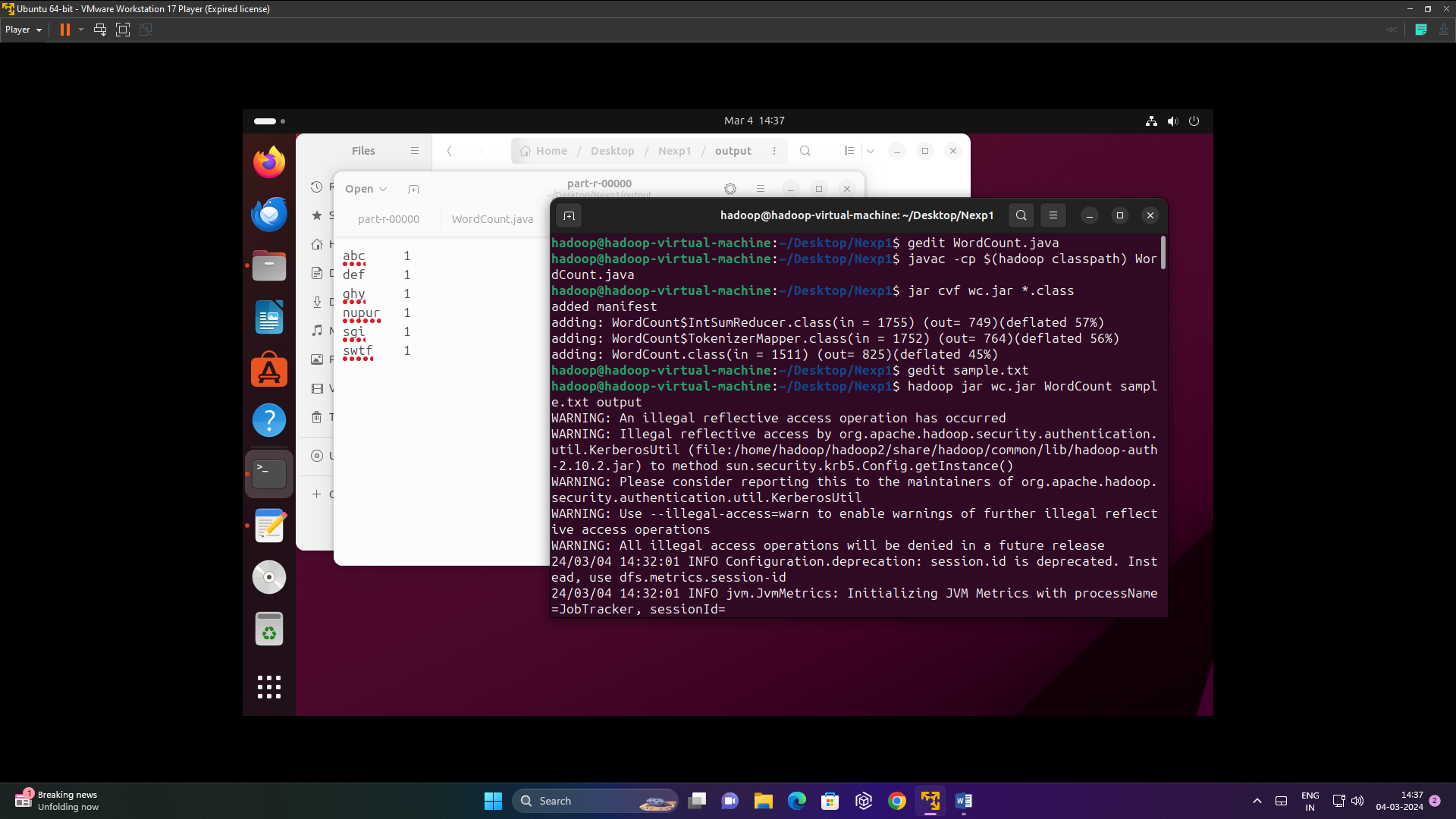
To develop a MapReduce program using Apache Hadoop to efficiently calculate the frequency of a given word within a specified file. MapReduce is a programming model specifically designed for processing large volumes of data in a distributed manner. By leveraging the parallel processing capabilities of Hadoop, this experiment seeks to demonstrate an effective approach to analyzing text data and derive useful insights regarding word frequency.

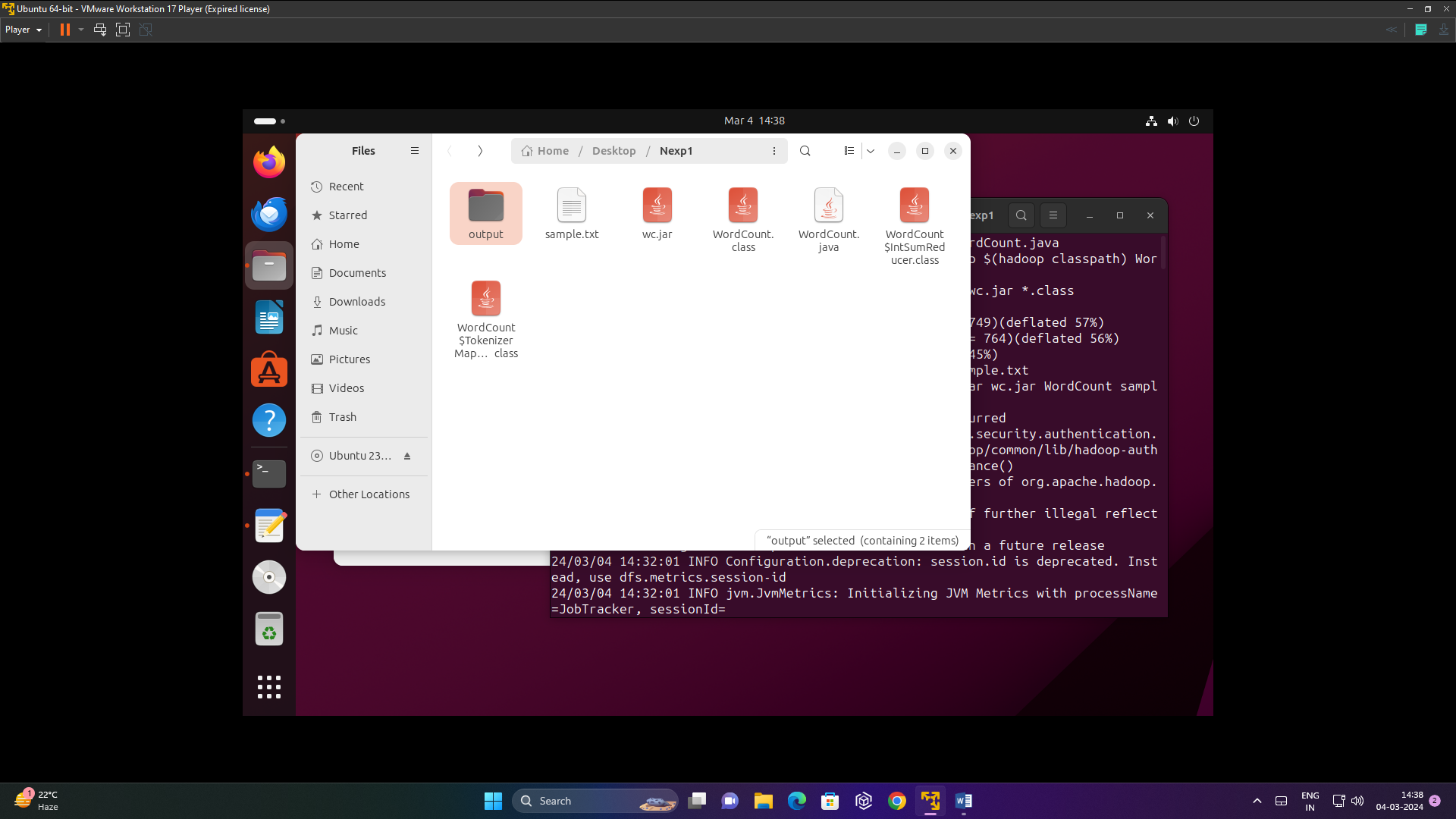
Word frequency analysis is a fundamental task in natural language processing, which involves determining the occurrence count of each word within a given document or corpus. Apache Hadoop provides a scalable framework for distributed computing, facilitating the implementation of parallel processing algorithms such as MapReduce. This experiment utilizes the MapReduce paradigm to distribute the workload across multiple nodes in a Hadoop cluster, enabling faster processing and analysis of text data.

The primary objective of this experiment is to develop a MapReduce program using Apache Hadoop to calculate the frequency of a specified word within a given text file. By distributing the computation across multiple nodes in a Hadoop cluster, the experiment aims to demonstrate the scalability and efficiency of MapReduce in processing large volumes of text data.

The experiment demonstrates the effectiveness of using Apache Hadoop and the MapReduce paradigm for calculating word frequency in large text datasets. By distributing the computation across multiple nodes, Hadoop enables efficient processing of big data, making it suitable for a wide range of analytical tasks in various domains.

**Steps:**





**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;

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");

job.setJarByClass(WordCount.class);

job.setMapperClass(TokenizerMapper.class);

job.setCombinerClass(IntSumReducer.class);

job.setReducerClass(IntSumReducer.class);

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(IntWritable.class);

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

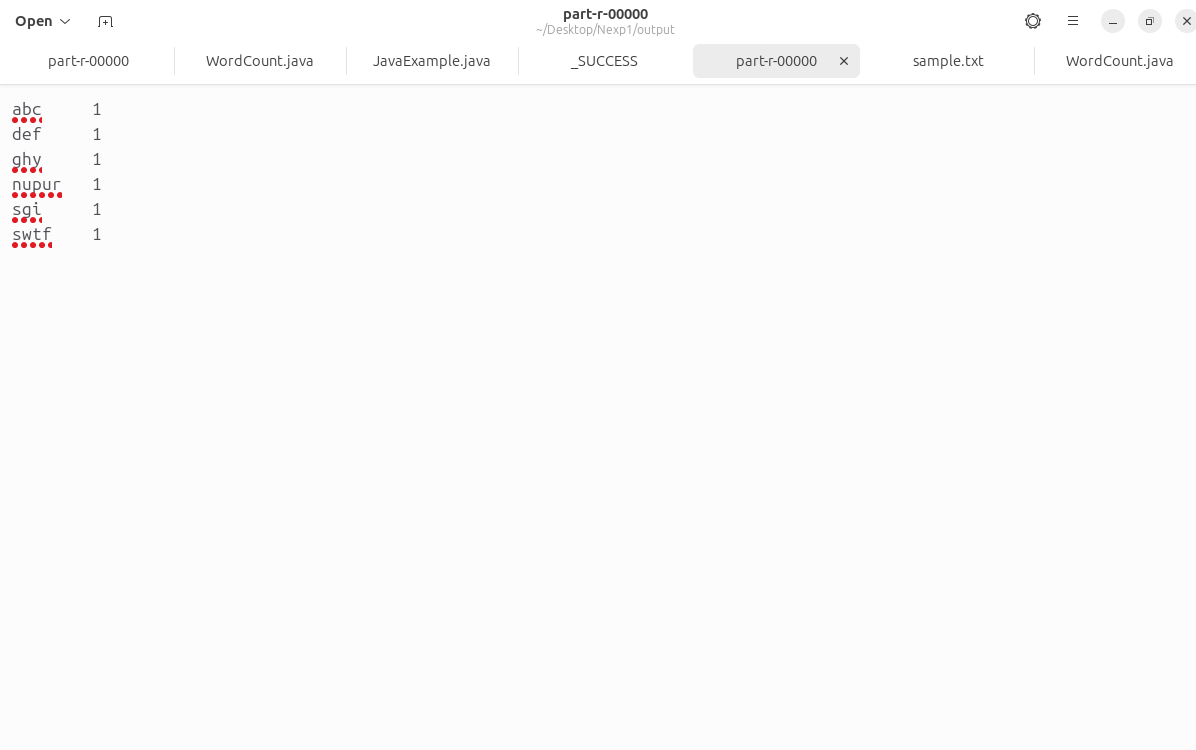
FileOutputFormat.setOutputPath(job, new Path(args[1]));

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

}

}

**Output:**



**PRACTICAL 3**

**Aim:** Develop a map reduction program to find the grades of students.

**Theory:**

To develop a MapReduce program using Apache Hadoop to efficiently calculate the grades of students. MapReduce is a programming model and processing framework designed to handle large scale data processing tasks distributedly. The input data for our program will consist of records representing student information, such as student ID, name, scores in various subjects, and possibly other relevant details. The objective is to calculate the final grades for each student based on their scores and any predefined grading criteria.

The MapReduce program will consist of two main phases: the Map phase and the Reduce phase.

1. Map Phase: In the Map phase, each mapper task will process a portion of the input data independently.

The mapper will parse the input records, extract the relevant information (such as student ID and scores), and perform any necessary calculations or transformations.

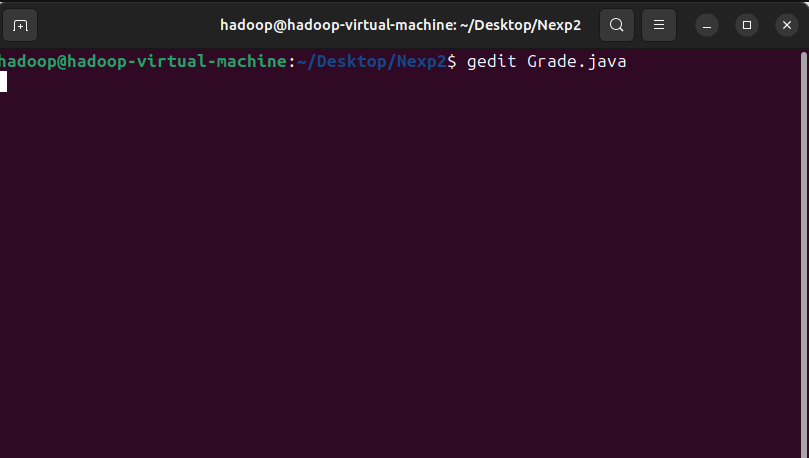
For each input record, the mapper will output key-value pairs, where the key represents the student ID and the value includes relevant information for grade calculation (e.g., subject scores).

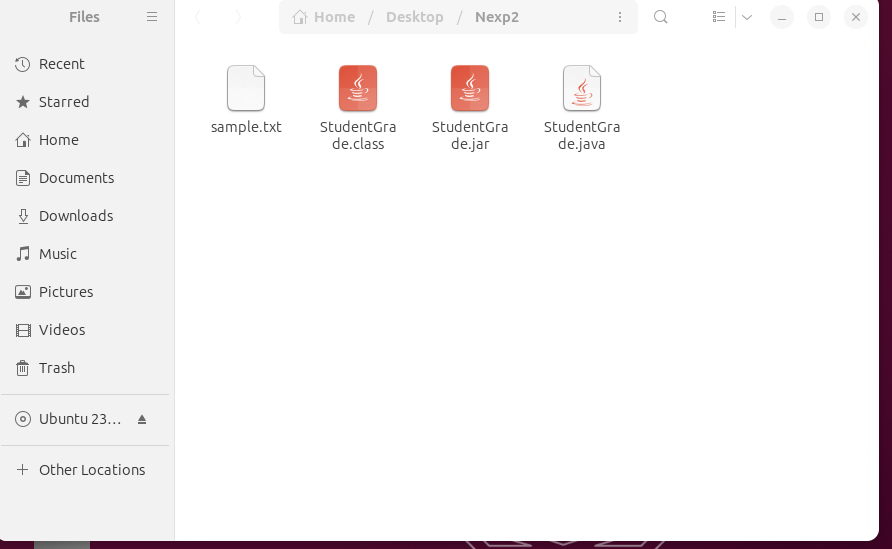
1. Reduce Phase: In the Reduce phase, the output of the Map phase will be aggregated and processed to calculate the final grades for each student. The reducer tasks will receive the key-value pairs generated by the mappers, grouped by student ID.

For each student, the reducer will compute the final grade based on the provided scores and any predefined grading criteria.

The final output of the Reduce phase will be key-value pairs containing the student ID and their corresponding final grade.

**Steps:**





**Code:**

import java.util.Scanner;

public class StudentGrade {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

int numStudents = 5;

String[] names = new String[numStudents];

int[] rollNumbers = new int[numStudents];

String[] subjects = new String[numStudents];

char[] grades = new char[numStudents];

for (int i = 0; i < numStudents; i++) {

System.out.println("Enter details for student " + (i + 1));

System.out.print("Name: ");

names[i] = scanner.nextLine();

System.out.print("Roll Number: ");

rollNumbers[i] = scanner.nextInt();

scanner.nextLine(); // Consume newline character

System.out.print("Subject: ");

subjects[i] = scanner.nextLine();

System.out.print("Grade: ");

grades[i] = scanner.nextLine().charAt(0);

}

// Print details for all students

System.out.println("Student Details:");

for (int i = 0; i < numStudents; i++) {

System.out.println("Student " + (i + 1) + ":");

System.out.println("Name: " + names[i]);

System.out.println("Roll Number: " + rollNumbers[i]);

System.out.println("Subject: " + subjects[i]);

System.out.println("Grade: " + grades[i]);

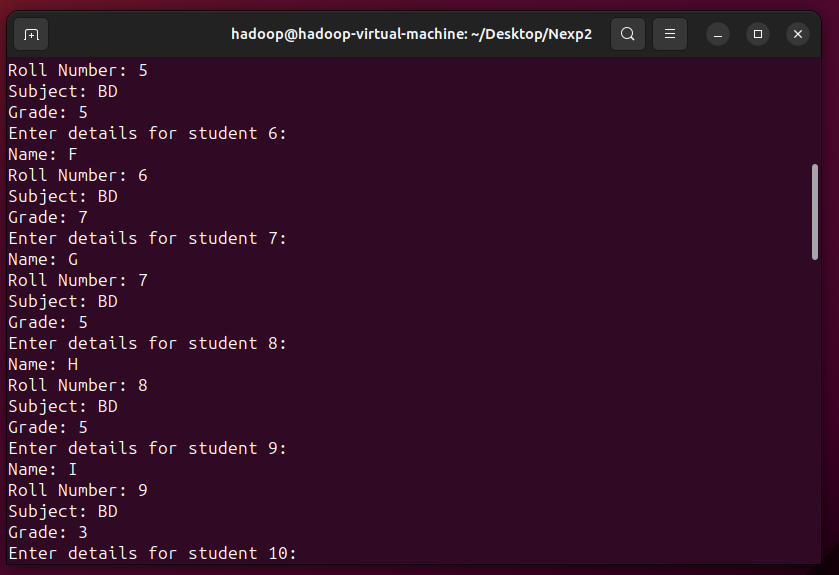
System.out.println();

}

}

}

**Output:**



**PRACTICAL 4**

**PRACTICAL 4**

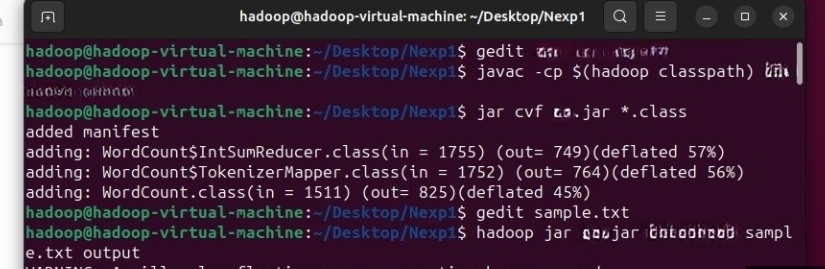
**Aim:** Develop a program to calculate the maximum recorded temperature year wise for the weather data set in Pig Latin.

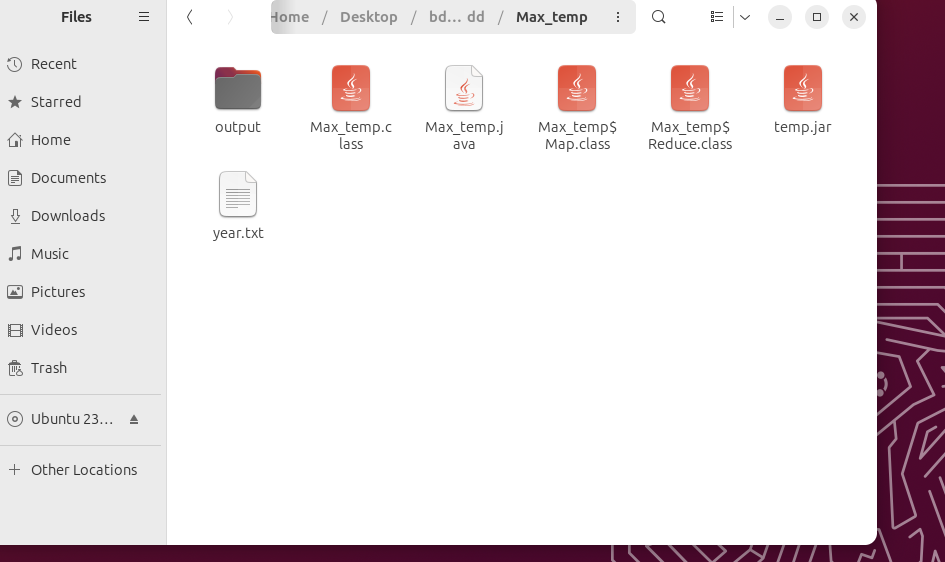
**Theory:**

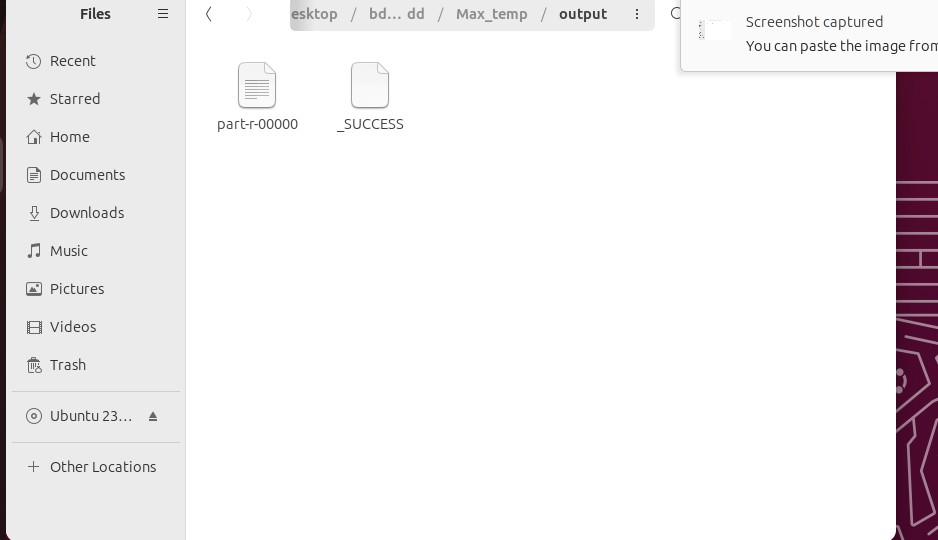
Apache Hadoop is a powerful framework for processing and analyzing large datasets in a distributed manner across clusters of computers using simple programming models. To calculate the maximum recorded temperature year wise in a weather dataset using Hadoop, we can follow a MapReduce paradigm.

1. **Input Data Format:** The weather dataset should be stored in a format suitable for Hadoop, such as Hadoop Distributed File System (HDFS) or any other Hadoop compatible file system. Each line in the dataset represents a weather record, typically containing fields like date, temperature, humidity, etc.
2. **Mapper Function**: The mapper function reads each weather record and extracts the year and temperature information. It emits key-value pairs where the key is the year and the value is the temperature.
3. **Reducer Function:** The reducer function receives key-value pairs grouped by year. For each year, it iterates through the temperature values and finds the maximum temperature. It emits the year along with the maximum temperature.
4. **Input Splitting**: Hadoop automatically divides the input data into manageable chunks called input splits, which are processed by individual mapper tasks.
5. **MapReduce Workflow:** Hadoop distributes the mapper tasks across the cluster, where each mapper processes a portion of the input data. The output of the mapper tasks is shuffled and sorted by key, then passed to the reducer tasks. Reducer tasks aggregate the intermediate results and compute the maximum temperature for each year. The final output is written to the output directory specified by the user.
6. **Handling Edge Cases**: Ensure handling of missing or invalid temperature values. Consider data skewness to optimize reducer performance.
7. **Output**: The final output will contain key-value pairs where the key is the year and the value is the maximum recorded temperature for that year.
8. **Execution:** The Hadoop job is submitted to the cluster using the appropriate commands or APIs provided by the Hadoop ecosystem.

**Steps:**

****





**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.LongWritable;

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.input.TextInputFormat;

import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

public class Max\_temp {

public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {

// Mapper

Text k = new Text();

@Override

public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {

String line = value.toString();

StringTokenizer tokenizer = new StringTokenizer(line, " ");

while (tokenizer.hasMoreTokens()) {

String year = tokenizer.nextToken();

k.set(year);

String temp = tokenizer.nextToken().trim();

int v = Integer.parseInt(temp);

context.write(k, new IntWritable(v));

}

}

}

// Reducer

public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {

@Override

public void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException, InterruptedException {

int maxtemp = 0;

for (IntWritable it : values) {

int temperature = it.get();

if (maxtemp < temperature) {

maxtemp = temperature;

}

}

context.write(key, new IntWritable(maxtemp));

}

}

// Driver

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

Configuration conf = new Configuration();

Job job = new Job(conf, "Max\_temp");

job.setJarByClass(Max\_temp.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]);

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

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

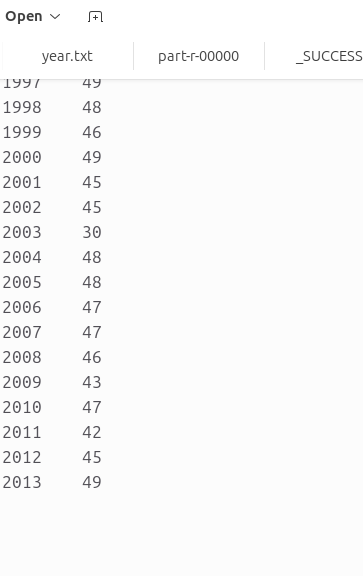
outputPath.getFileSystem(conf).delete(outputPath, true);

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

}

}

**Output:**



**PRACTICAL 5**

**Aim:** Develop a map reduce program to implement matrix multiplication.

**Theory:**

Matrix multiplication is a fundamental operation in many computational tasks, and implementing it in a distributed computing framework like Apache Hadoop can provide significant benefits in terms of scalability and performance.

Matrix multiplication involves multiplying two matrices to produce another matrix. Given two matrices A (of dimensions m x n) and B (of dimensions n x p), the resulting matrix C (of dimensions m x p) is computed as follows:

C[i][j] = Σ(A[i][k] \* B[k][j]) for k = 1 to n, where 0 <= i< m and 0 <= j < p.

To implement matrix multiplication in Apache Hadoop using MapReduce, we can follow these steps:

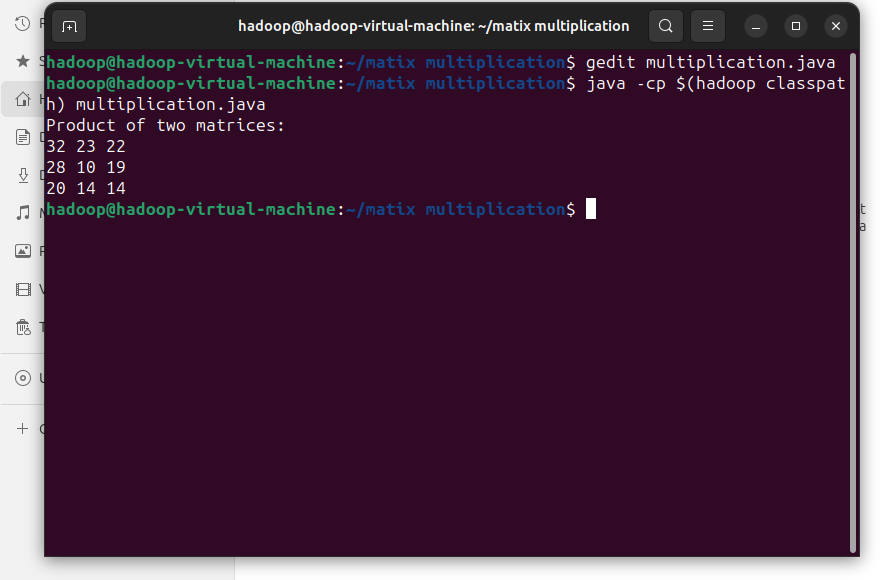
1. Input Data Representation: Represent each input matrix as a set of key-value pairs, where the key represents the row/column index, and the value represents the matrix element. For example, for matrix A, the key would be (i, j) where i is the row index and j is the column index, and the value would be the corresponding matrix element.
2. Map Function: In the map function, we emit intermediate key-value pairs for each element of the input matrices.

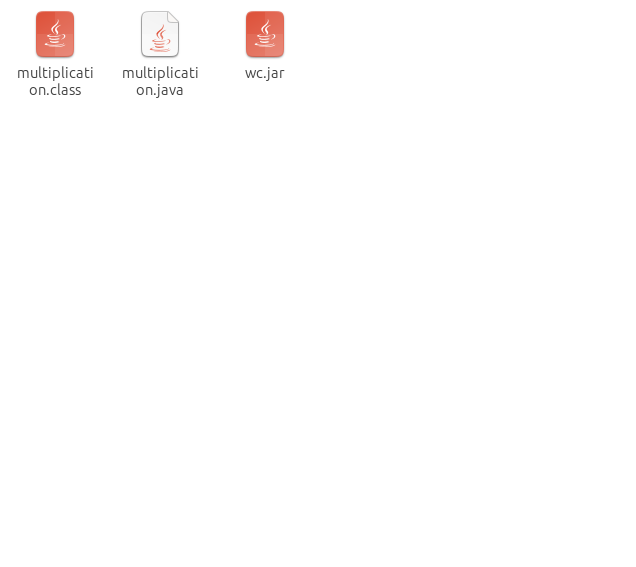
For matrix A, emit (i, (A, j, A[i][j])) for each element.

For matrix B, emit (j, (B, i, B[j][i])) for each element.

1. Partitioning and Shuffle: The Hadoop framework shuffles and sorts the intermediate key-value pairs based on keys. All pairs with the same key (i.e., the same row/column index) will be grouped and sent to the same reducer.
2. Reduce Function: In the reduce function, compute the product of corresponding elements from the intermediate pairs to get the resulting matrix elements. For each key (i.e., row/column index), multiply corresponding elements from matrix A and matrix B and sum them up to get the result for C[i][j]. Emit the final result as (i, j, C[i][j]).
3. Output Data: Collect all the output key-value pairs from reducers to get the final result matrix C.

**Steps:**





**Code:**

import java.io.IOException;

import java.util.\*;

import java.util.AbstractMap.SimpleEntry;

import java.util.Map.Entry;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.conf.\*;

import org.apache.hadoop.io.\*;

import org.apache.hadoop.mapreduce.\*;

import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;

import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;

import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

public class TwoStepMatrixMultiplication {

public static class Map extends Mapper<LongWritable, Text, Text, Text> {

public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {

String line = value.toString();

String[] indicesAndValue = line.split(",");

Text outputKey = new Text();

Text outputValue = new Text();

if (indicesAndValue[0].equals("A")) {

outputKey.set(indicesAndValue[2]);

outputValue.set("A," + indicesAndValue[1] + "," + indicesAndValue[3]);

context.write(outputKey, outputValue);

} else {

outputKey.set(indicesAndValue[1]);

outputValue.set("B," + indicesAndValue[2] + "," + indicesAndValue[3]);

context.write(outputKey, outputValue);

}

}

}

public static class Reduce extends Reducer<Text, Text, Text, Text> {

public void reduce(Text key, Iterable<Text> values, Context context) throws IOException, InterruptedException {

String[] value;

ArrayList<Entry<Integer, Float>> listA = new ArrayList<Entry<Integer, Float>>();

ArrayList<Entry<Integer, Float>> listB = new ArrayList<Entry<Integer, Float>>();

for (Text val : values) {

value = val.toString().split(",");

if (value[0].equals("A")) {

listA.add(new SimpleEntry<Integer, Float>(Integer.parseInt(value[1]), Float.parseFloat(value[2])));

} else {

listB.add(new SimpleEntry<Integer, Float>(Integer.parseInt(value[1]), Float.parseFloat(value[2])));

}

}

String i;

float a\_ij;

String k;

float b\_jk;

Text outputValue = new Text();

for (Entry<Integer, Float> a : listA) {

i = Integer.toString(a.getKey());

a\_ij = a.getValue();

for (Entry<Integer, Float> b : listB) {

k = Integer.toString(b.getKey());

b\_jk = b.getValue();

outputValue.set(i + "," + k + "," + Float.toString(a\_ij \* b\_jk));

context.write(null, outputValue);

}

}

}

}

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

Configuration conf = new Configuration();

Job job = new Job(conf, "MatrixMatrixMultiplicationTwoSteps");

job.setJarByClass(TwoStepMatrixMultiplication.class);

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(Text.class);

job.setMapperClass(Map.class);

job.setReducerClass(Reduce.class);

job.setInputFormatClass(TextInputFormat.class);

job.setOutputFormatClass(TextOutputFormat.class);

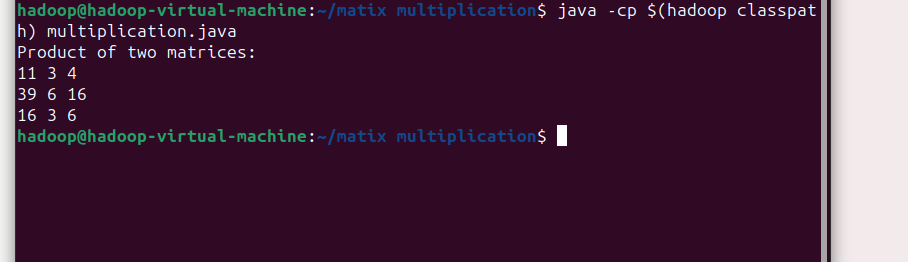
FileInputFormat.addInputPath(job, new Path("hdfs://127.0.0.1:9000/matrixin"));

FileOutputFormat.setOutputPath(job, new Path("hdfs://127.0.0.1:9000/matrixout"));

job.waitForCompletion(true);

}

}



**PRACTICAL 6**

**Aim:** Develop a map reduce program to find the maximum electrical consumption in each year given electrical consumption for each month in each year.

**Theory:**

1. Input Data: The input data will consist of records containing information about electrical consumption for each month in each year. Each record will include fields such as year, month, and consumption value.

2. Mapper Function: The Mapper function will parse each input record and emit keyvalue pairs where the key is the year, and the value is the consumption for that month.

* Example input record: `(year, month, consumption)`
* Mapper emits: `(year, consumption)`

3. Partitioner: Optionally, a custom partitioner can be implemented to ensure that all records with the same key (year) are sent to the same reducer.

4. Sorting: Hadoop's shuffle and sort phase will automatically group the keyvalue pairs by key (year) and sort them in ascending order of the key.

5. Reducer Function:The Reducer function will receive all the consumption values for a particular year.

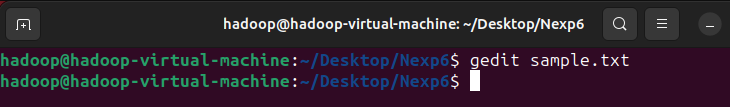
* It will then iterate through these values to find the maximum consumption for that year.
* Finally, it will emit the maximum consumption for each year.
* Example input to Reducer: `(year, [consumption1, consumption2, ...])`
* Reducer emits: `(year, max\_consumption)`

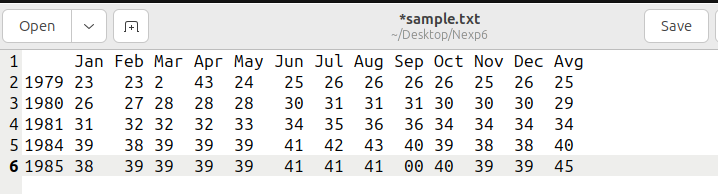
6. Output: The output of the MapReduce job will consist of keyvalue pairs where the key is the year, and the value is the maximum electrical consumption for that year.

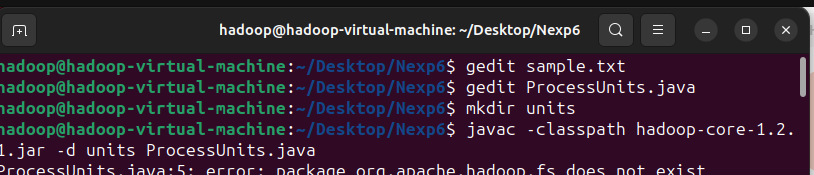
7. Execution:The MapReduce job will be submitted to the Hadoop cluster for execution.

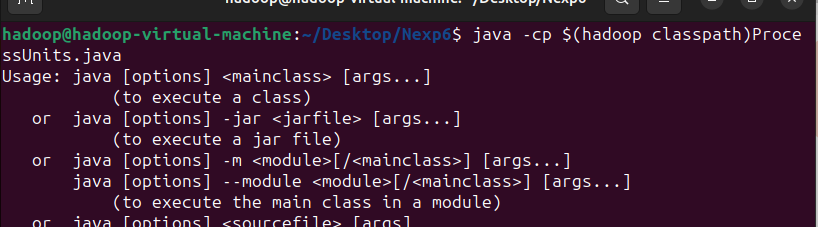
* Hadoop will automatically distribute the input data across the cluster, execute the Mapper and Reducer tasks in parallel, and handle the shuffle and sort phase.
* The output will be written to HDFS (Hadoop Distributed File System) or any specified output location.

**Steps:**

****

****

****

****

****

****

**Code:**

package hadoop;

import java.util.\*;

import java.io.IOException;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.conf.\*;

import org.apache.hadoop.io.\*;

import org.apache.hadoop.mapred.\*;

import org.apache.hadoop.util.\*;

public class ProcessUnits {

// Mapper class

public static class E\_EMapper extends MapReduceBase implements Mapper<LongWritable, Text, Text, IntWritable> {

// Map function

public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {

String line = value.toString();

String lasttoken = null;

StringTokenizer s = new StringTokenizer(line, "\t");

String year = s.nextToken();

while (s.hasMoreTokens()) {

lasttoken = s.nextToken();

}

int avgprice = Integer.parseInt(lasttoken);

output.collect(new Text(year), new IntWritable(avgprice));

}

}

public static class E\_EReduce extends MapReduceBase implements Reducer<Text, IntWritable, Text, IntWritable> {

public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {

int maxavg = 30;

int val = Integer.MIN\_VALUE;

while (values.hasNext()) {

if ((val = values.next().get()) > maxavg) {

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

}

}

}

}

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

JobConf conf = new JobConf(ProcessUnits.class);

conf.setJobName("max\_eletricityunits");

conf.setOutputKeyClass(Text.class);

conf.setOutputValueClass(IntWritable.class);

conf.setMapperClass(E\_EMapper.class);

conf.setCombinerClass(E\_EReduce.class);

conf.setReducerClass(E\_EReduce.class);

conf.setInputFormat(TextInputFormat.class);

conf.setOutputFormat(TextOutputFormat.class);

FileInputFormat.setInputPaths(conf, new Path(args[0]));

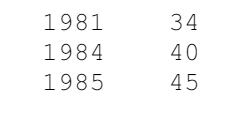
FileOutputFormat.setOutputPath(conf, new Path(args[1]));

JobClient.runJob(conf);

}

}

**Output:**

****

**PRACTICAL 7**

**Aim:** Develop a map reduce program to analyze the weather data set and print whether the day is shiny or cool.

**Theory:**

1. Input Data: The input dataset will contain weather information such as temperature, humidity, wind speed, etc., recorded over a period of time, typically on a daily basis. Each record in the dataset represents the weather conditions for a particular day.

2. Mapper Function:Mapper function reads each record from the input dataset.

* It extracts relevant information such as temperature and weather conditions for each day.
* Based on certain criteria (e.g., temperature range), it categorizes each day as either shiny or cool.
* It emits key-value pairs where the key is the day (or date) and the value is the category (shiny or cool).

3. Reducer Function: The Reducer function receives key-value pairs emitted by the Mapper function.

* It aggregates the data based on the key, which is the day (or date).
* For each day, it determines the majority category (shiny or cool) based on the values associated with that key.
* It outputs the day along with the corresponding majority category.

4. Output: The output of the MapReduce program will be a list of days (or dates) along with their corresponding weather category (shiny or cool).

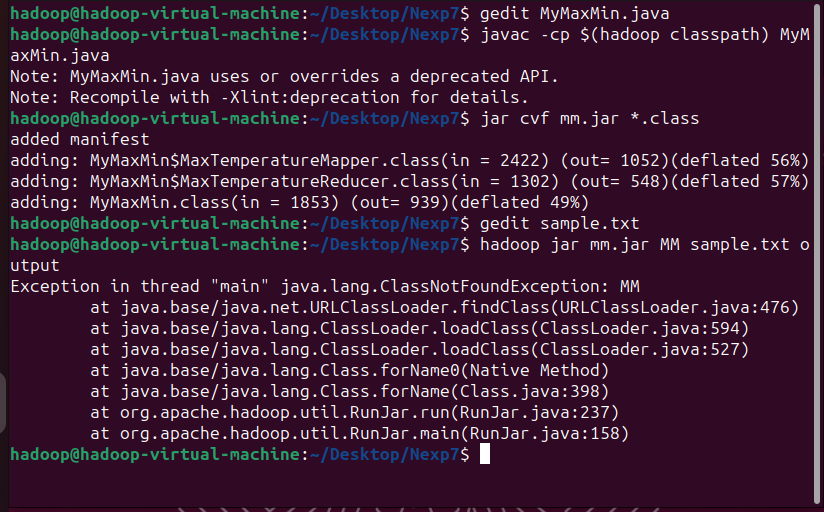
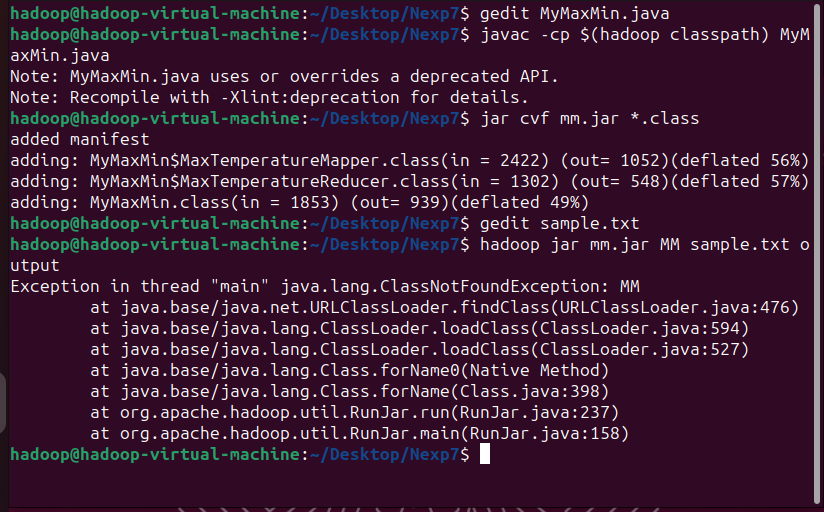
5. Execution: The MapReduce program is executed on a Hadoop cluster. Input data is distributed across the cluster's nodes. Mapper tasks run in parallel across the nodes, processing subsets of the input data.

* + Intermediate key-value pairs generated by the mappers are shuffled and sorted.
  + Reducer tasks receive shuffled data, aggregate it, and produce the final output.

6. Parameters and Thresholds: Thresholds for categorizing days as shiny or cool can be set based on temperature ranges, humidity levels, or other relevant factors.These thresholds can be configurable parameters in the MapReduce program, allowing users to adjust them as needed.

7. Handling Edge Cases:The MapReduce program should handle edge cases such as missing or invalid data gracefully. It should also consider scenarios where a day's weather conditions might fall into neither the shiny nor cool category.

**Steps:**

****

**Code:**

// importing Libraries

import java.io.IOException;

import java.util.Iterator;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.LongWritable;

import org.apache.hadoop.io.Text;

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;

import org.apache.hadoop.mapreduce.Job;

import org.apache.hadoop.mapreduce.Mapper;

import org.apache.hadoop.mapreduce.Reducer;

import org.apache.hadoop.conf.Configuration;

public class MyMaxMin {

public static class MaxTemperatureMapper extends Mapper<LongWritable, Text, Text, Text> {

public static final int MISSING = 9999;

@Override

public void map(LongWritable arg0, Text value, Context context) throws IOException, InterruptedException {

String line = value.toString();

if (!(line.length() == 0)) {

String date = line.substring(6, 14);

float tempMax = Float.parseFloat(line.substring(39, 45).trim());

float tempMin = Float.parseFloat(line.substring(47, 53).trim());

if (tempMax > 30.0) {

context.write(new Text("The Day is Hot Day :" + date), new Text(String.valueOf(tempMax)));

}

if (tempMin < 15) {

context.write(new Text("The Day is Cold Day :" + date), new Text(String.valueOf(tempMin)));

}

}

}

}

public static class MaxTemperatureReducer extends Reducer<Text, Text, Text, Text> {

public void reduce(Text key, Iterator<Text> values, Context context) throws IOException, InterruptedException {

String temperature = values.next().toString();

context.write(key, new Text(temperature));

}

}

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

Configuration conf = new Configuration();

Job job = new Job(conf, "weather example");

job.setJarByClass(MyMaxMin.class);

job.setMapOutputKeyClass(Text.class);

job.setMapOutputValueClass(Text.class);

job.setMapperClass(MaxTemperatureMapper.class);

job.setReducerClass(MaxTemperatureReducer.class);

job.setInputFormatClass(TextInputFormat.class);

job.setOutputFormatClass(TextOutputFormat.class);

Path outputPath = new Path(args[1]);

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

FileOutputFormat.setOutputPath(job, outputPath);

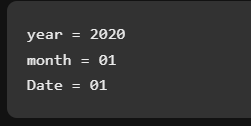
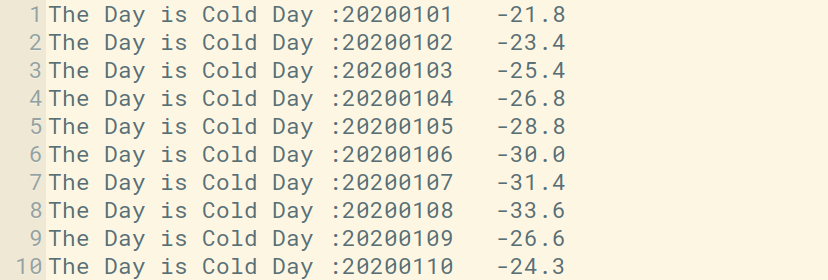
outputPath.getFileSystem(conf).delete(outputPath, true);

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

}

}

**Output:**



**PRACTICAL 8**

**Aim:** Develop a map-reduce program to find the tags associated with each movie by analyzing movie lens data.

**Theory:**

1. Data Understanding: need to understand the structure of the MovieLens dataset. It typically consists of several CSV or JSON files containing information about movies, ratings, tags, etc. In particular, you need to focus on the files containing movie data and tags associated with each movie.

2. MapReduce Workflow: Mapper Phase: In this phase, each mapper task reads a portion of the input data and processes it. For this program, each mapper will take as input a portion of the movie data, extract movie IDs and associated tags, and emit key-value pairs where the movie ID is the key and the tag(s) associated with the movie are the value(s).

Reducer Phase: In this phase, all key-value pairs emitted by the mappers are shuffled and sorted by key (movie ID). Then, each reducer task processes all values associated with a particular movie ID and aggregates them to find the complete list of tags associated with that movie.

3. Mapper Function: mapper function will parse each line of input data and extract the movie ID and its associated tags. It will emit key-value pairs where the movie ID is the key and the tags associated with that movie are the values.

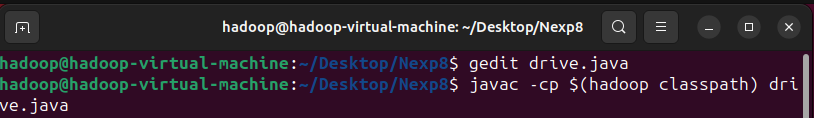
4. Reducer Function: The reducer function will receive key-value pairs where the key is a movie ID and the values are lists of tags associated with that movie. It will iterate through the list of tags for each movie, eliminating duplicates if any, and aggregate them into a single list.

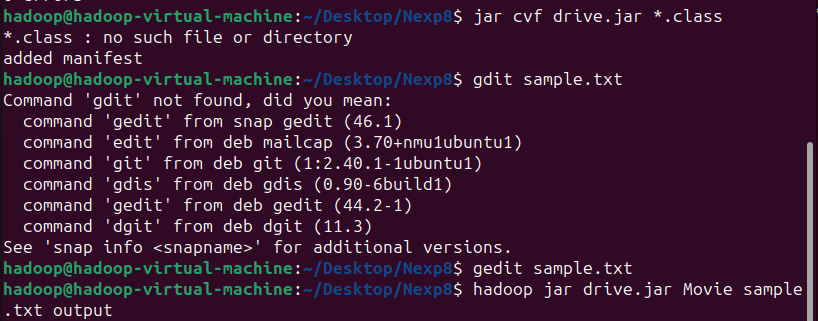
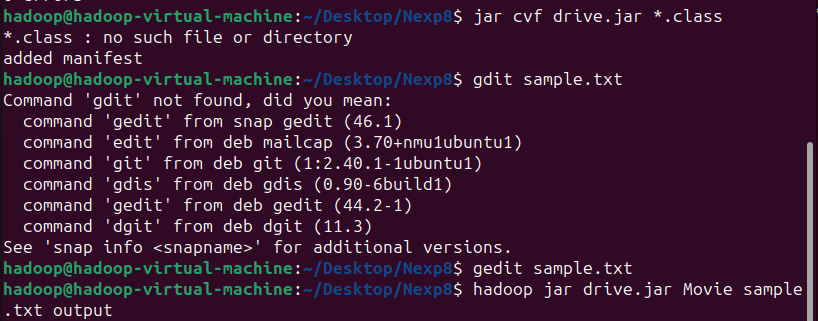
5. Output: The output of the MapReduce job will be key-value pairs where the key is a movie ID and the value is a list of tags associated with that movie.

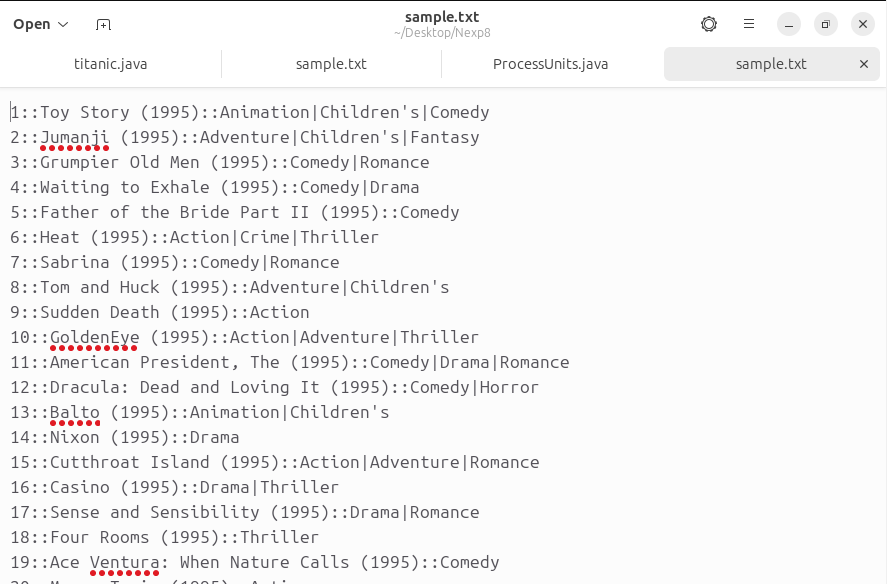
6. Execution: The MapReduce job will be executed on a Hadoop cluster, where the input data will be distributed across multiple nodes for parallel processing. Mappers and reducers will run concurrently on different nodes, processing their respective portions of the input data.

7. Optimizations: To optimize performance, you can consider techniques such as combiners, which perform local aggregation in the mapper phase to reduce the amount of data shuffled across the network, partitioning the data based on movie ID can improve the efficiency of the shuffle phase.

**Steps:**

****

****

****

**Code:**

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.LongWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Job;

import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;

import org.apache.hadoop.mapreduce.lib.input.MultipleInputs;

import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;

import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import org.apache.hadoop.util.GenericOptionsParser;

public class Driver1 {

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

Path firstPath = new Path(args[0]);

Path secondPath = new Path(args[1]);

Path outputPath1 = new Path(args[2]);

Path outputPath2 = new Path(args[3]);

Configuration conf = new Configuration();

Job job = Job.getInstance(conf, "Most Viewed Movies");

job.setJarByClass(Driver1.class);

job.setMapOutputKeyClass(LongWritable.class);

job.setMapOutputValueClass(Text.class);

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(LongWritable.class);

MultipleInputs.addInputPath(job, firstPath, TextInputFormat.class, movieDataMapper.class);

MultipleInputs.addInputPath(job, secondPath, TextInputFormat.class, ratingDataMapper.class);

job.setReducerClass(dataReducer.class);

FileOutputFormat.setOutputPath(job, outputPath1);

job.waitForCompletion(true);

Job job1 = Job.getInstance(conf, "Most Viewed Movies2");

job1.setJarByClass(Driver1.class);

job1.setMapperClass(topTenMapper.class);

job1.setReducerClass(topTenReducer.class);

job1.setMapOutputKeyClass(Text.class);

job1.setMapOutputValueClass(LongWritable.class);

job1.setOutputKeyClass(LongWritable.class);

job1.setOutputValueClass(Text.class);

FileInputFormat.addInputPath(job1, outputPath1);

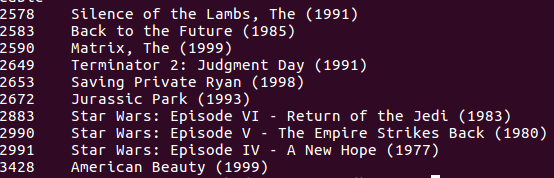
FileOutputFormat.setOutputPath(job1, outputPath2);

job1.waitForCompletion(true);

}

}

**Output:**



**PRACTICAL 9**

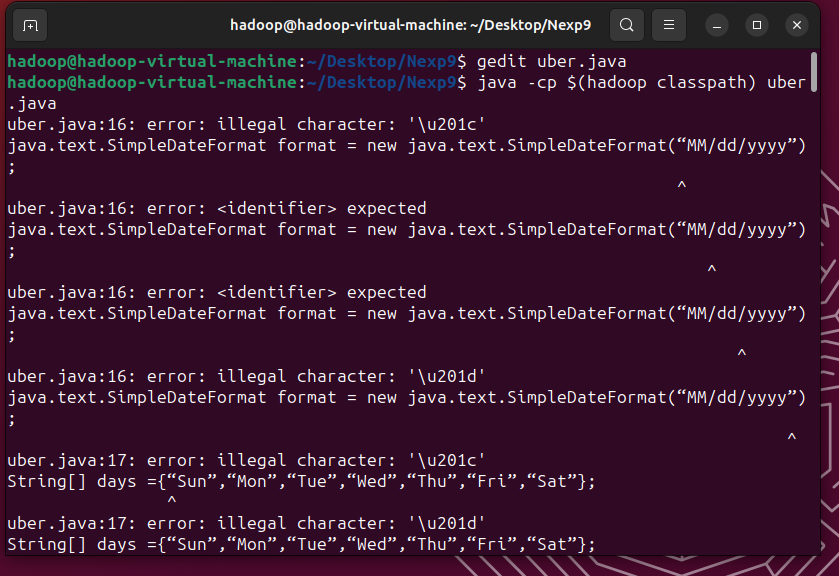
**Aim:** Develop a map reduce program to analyze Uber data set to find the days on which each basement has more trips using the following data set. The Uber data set consists of four columns:Dispatching, base, no. date active, vehicle trips.

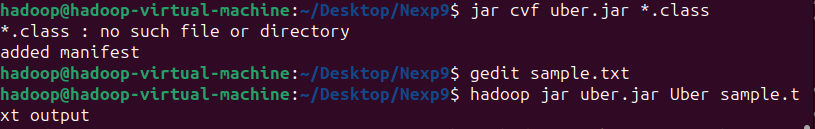
**Theory:**

1. Input Data: The input data consists of Uber records with four columns: Dispatching, base, date active, and vehicle trips.
2. Mapper: The Mapper reads each line from the input data. It extracts the base and date information from each record. The key emitted by the Mapper would be a composite key consisting of the base and date, while the value would be the number of trips on that day for that base.
3. Shuffle and Sort: The MapReduce framework shuffles and sorts the Mapper output based on the composite key (base and date) to group records with the same base together.
4. Reducer: Reducer receives groups of records, each group representing trips for a specific base on a particular date.For each group, the Reducer calculates the total number of trips.
5. Output: The output of the Reducer is written to the Hadoop Distributed File System (HDFS) or any other storage system as required. The output format could be base along with the dates with the maximum number of trips.
6. Execution: The MapReduce job is executed on a Hadoop cluster. The Hadoop framework distributes the task among Mapper and Reducer nodes, ensuring parallel processing.
7. Optimizations: Combiners can be used to optimize the amount of data shuffled between Mappers and Reducers, especially if there's a significant amount of data for each base.

Use of appropriate data types and serialization techniques can improve performance and reduce storage requirements.

**Steps:**

****

****

**Code:**

import java.io.IOException;

import java.text.ParseException;

import java.util.Date;

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;

public class Uber1 {

public static class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable> {

java.text.SimpleDateFormat format = new java.text.SimpleDateFormat("MM/dd/yyyy");

String[] days = {"Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"};

private Text basement = new Text();

Date date = null;

private int trips;

public void map(Object key, Text value, Context context) throws IOException, InterruptedException {

String line = value.toString();

String[] splits = line.split(",");

basement.set(splits[0]);

try {

date = format.parse(splits[1]);

} catch (ParseException e) {

// TODO Autogenerated catch block

e.printStackTrace();

}

trips = Integer.parseInt(splits[3]);

String keys = basement.toString() + " " + days[date.getDay()];

context.write(new Text(keys), new IntWritable(trips));

}

}

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, "Uber1");

job.setJarByClass(Uber1.class);

job.setMapperClass(TokenizerMapper.class);

job.setCombinerClass(IntSumReducer.class);

job.setReducerClass(IntSumReducer.class);

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(IntWritable.class);

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

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

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

}

}

**Output:**



**PRACTICAL 10**

**Aim:** Develop a map reduce program to analyze titanic dataset to find the average age of the people (both male and female) who died in the tragedy. How many people survived in each class.

**Theory:**

1. Understanding the Dataset: Before diving into the MapReduce implementation, it's essential to understand the structure and format of the Titanic dataset. It typically contains columns such as PassengerId, Survived, Pclass, Name, Sex, Age, etc. For this analysis, we are particularly interested in the 'Survived', 'Sex', 'Age', and 'Pclass' columns.

2. MapReduce Workflow: Mapper: Input: Each mapper receives a line of input from the dataset.

* Task: Extract necessary information from each line, such as 'Survived', 'Sex', 'Age', and 'Pclass'.
* Output: Emit key-value pairs where the key is a composite key consisting of 'Sex' and 'Survived' or just 'Pclass', and the value is the age or count of survivors.

Reducer:Input: Receives key-value pairs emitted by the mappers.

Task: For finding the average age of people who died:Accumulate the sum of ages for each combination of 'Sex' and 'Survived'.Keep track of the count of occurrences for each combination.

Output: Emit the key-value pairs where the key indicates 'Sex' and 'Survived' or just 'Pclass', and the value is either the average age or the count of survivors.

3. Algorithm Steps:Mapper:

1. Read each line of the dataset.

2. Extract 'Sex', 'Age', 'Survived', and 'Pclass' from the line.

3. If 'Survived' is 0 (indicating the person died), emit key-value pairs with keys as 'Sex:Survived' and value as 'Age'.

4. Emit key-value pairs with keys as 'Pclass' and value as '1' for each passenger regardless of survival status.

Reducer:

1. Receive key-value pairs from all mappers.

2. For each unique key:If the key indicates 'Sex:Survived':

* Sum up the ages and count occurrences.
* If the key indicates 'Pclass':
* Count the occurrences.

3. Calculate the average age for each combination of 'Sex' and 'Survived'.

4. Emit the results for both tasks.

4. Output: The output will include the average age of people who died categorized by sex, and the count of survivors in each passenger class.

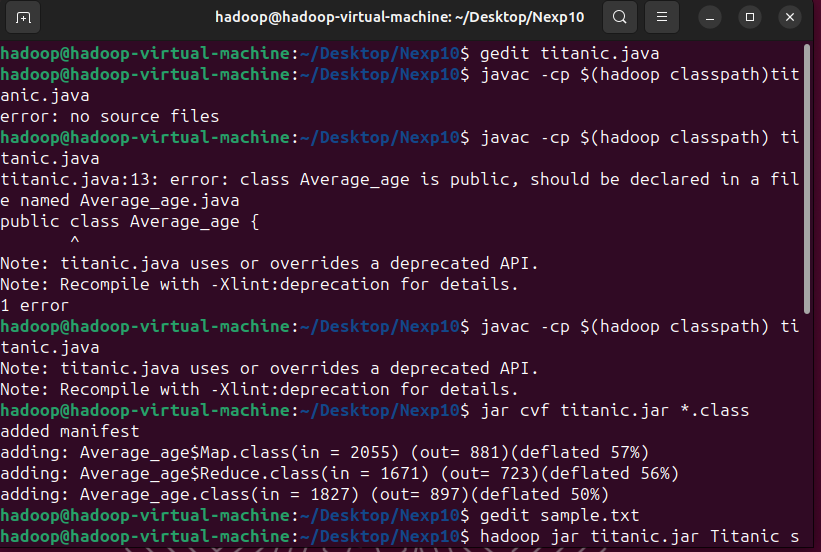
5. Hadoop Setup: Ensure that Hadoop is properly configured and the Titanic dataset is stored in Hadoop's distributed file system (HDFS).

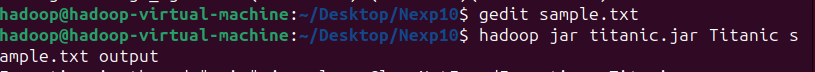
6. MapReduce Job Execution: Submit the MapReduce job to the Hadoop cluster, specifying the input dataset location, mapper, reducer, and output directory.

7. Result Analysis:

Once the MapReduce job is completed, analyze the output to get insights into the average age of people who died and the count of survivors in each class.

**Steps:**

****

****

**Code:**

// import libraries

import java.io.IOException;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.conf.\*;

import org.apache.hadoop.io.\*;

import org.apache.hadoop.mapreduce.\*;

import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;

import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;

import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

// Making a class with name Average\_age

public class Average\_age {

public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {

private Text gender = new Text();

private IntWritable age = new IntWritable();

public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException

{

String line = value.toString();

String str[] = line.split(", ");

if (str.length> 6) {

gender.set(str[4]);

if ((str[1].equals("0"))) {

if (str[5].matches("\\d+")) {

int i = Integer.parseInt(str[5]);

age.set(i);}}}

context.write(gender, age);}}

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;

int l = 0;

for (IntWritableval : values) {

l += 1;

sum += val.get();

}

sum = sum / l;

context.write(key, new IntWritable(sum));

}

}

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

{Configuration conf = new Configuration();

@SuppressWarnings("deprecation")

Job job = new Job(conf, "Averageage\_survived");

job.setJarByClass(Average\_age.class);

job.setMapOutputKeyClass(Text.class);

job.setMapOutputValueClass(IntWritable.class);

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(IntWritable.class);

job.setMapperClass(Map.class);

job.setReducerClass(Reduce.class);

job.setInputFormatClass(TextInputFormat.class);

job.setOutputFormatClass(TextOutputFormat.class);

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

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

Path out = new Path(args[1]);

out.getFileSystem(conf).delete(out);

job.waitForCompletion(true);

}

}

**Output:**

