**Experiment 1**

Aim: Install Apache Hadoop.

**Theory:**

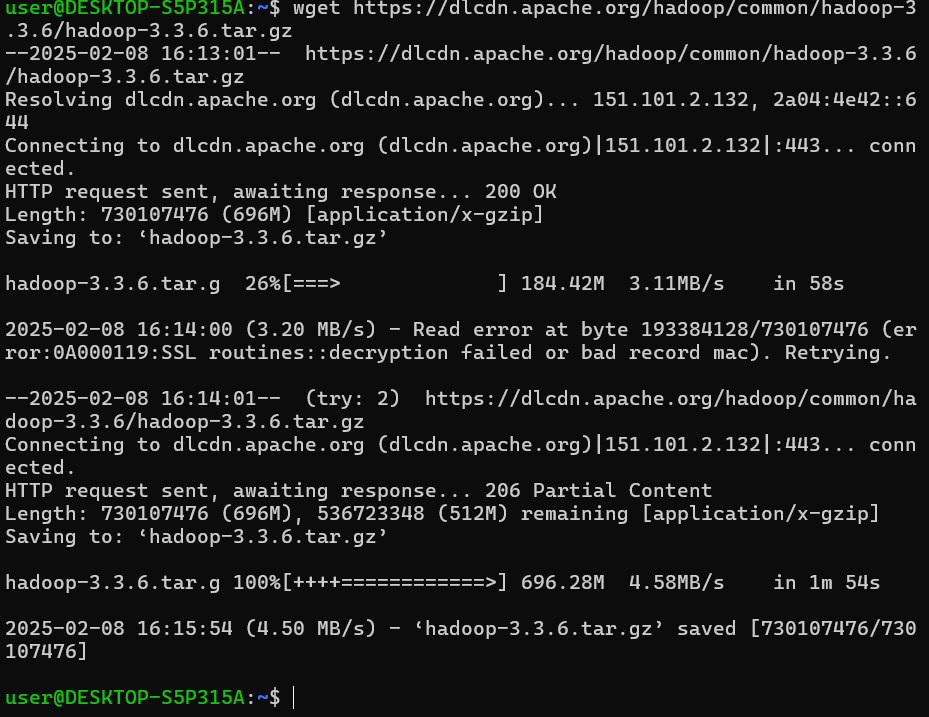
Apache Hadoop is a powerful framework designed for the distributed storage and processing of large datasets using clusters of commodity hardware. Its integration with Ubuntu provides a stable and efficient environment for big data processing. Here’s how Hadoop relates to Ubuntu:

1. **Installation** – Apache Hadoop can be installed on Ubuntu by downloading the official Hadoop binaries or using package managers like apt. Ubuntu’s robust package management system simplifies the setup of Hadoop clusters.
2. **Compatibility** – Hadoop runs smoothly on Ubuntu servers and desktops without major compatibility issues, allowing users to leverage Ubuntu’s stability and efficiency for big data processing.
3. **Resource Management** – Ubuntu offers various tools for managing system resources, which is crucial when running Hadoop clusters. Proper resource management ensures optimal performance and efficient utilization of cluster resources.
4. **Security** – Ubuntu provides strong security features, including firewall configurations, user permissions, and encryption, which help secure Hadoop clusters and protect stored and processed data.
5. **Maintenance** – With its regular updates and long-term support (LTS) releases, Ubuntu ensures the stability and security of Hadoop clusters over extended periods. Updates can be easily applied to both Ubuntu and Hadoop components for seamless operation.
6. **Community Support** – Both Hadoop and Ubuntu have active communities that offer extensive documentation, troubleshooting resources, and support, making it easier for users to resolve issues and stay updated with new developments.

By using Ubuntu as the operating system for Hadoop, users can take advantage of its reliability, security, and ease of maintenance to build scalable and efficient big data processing systems.

Steps:

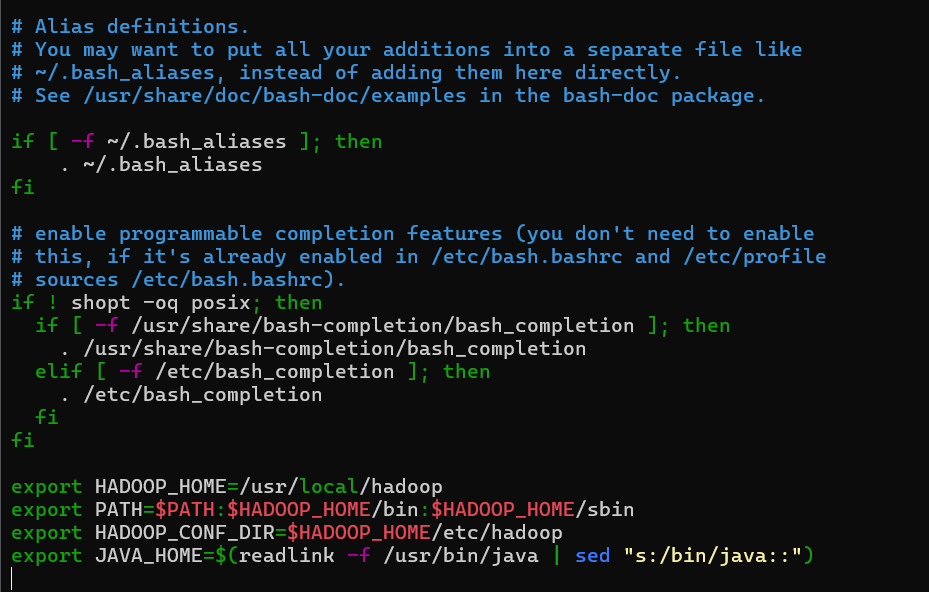
1. Install Hadoop in the virtual machine.



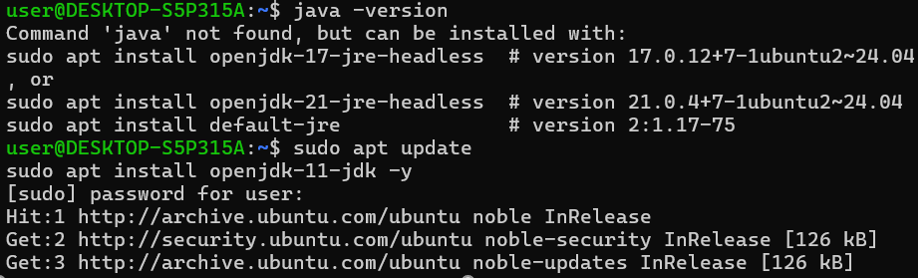
2. Unzip Hadoop



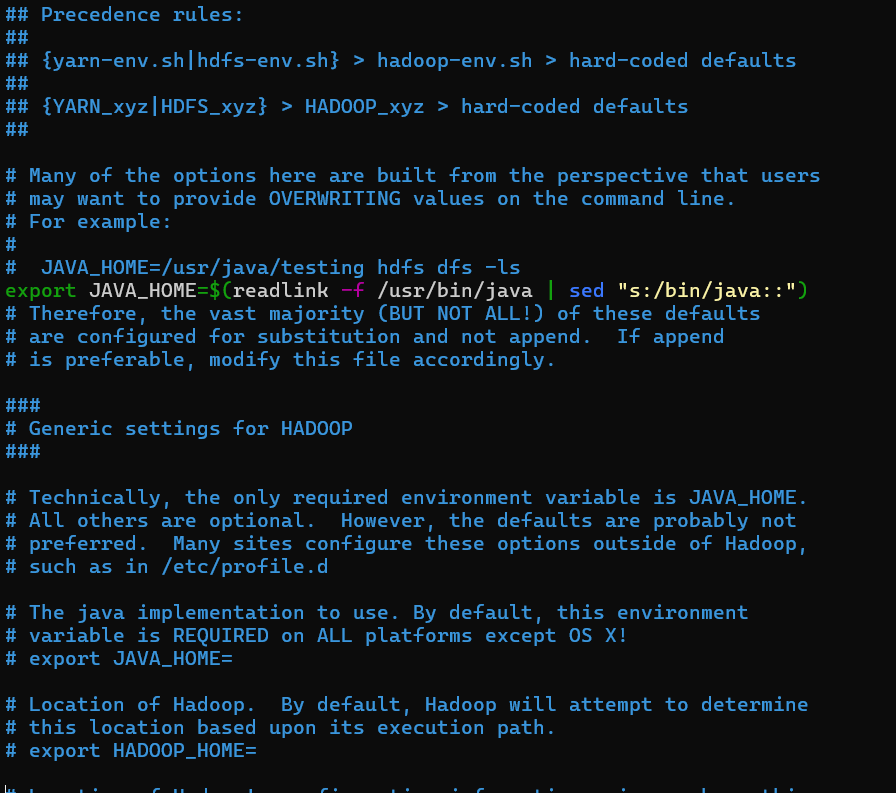
3. Setup environment variables



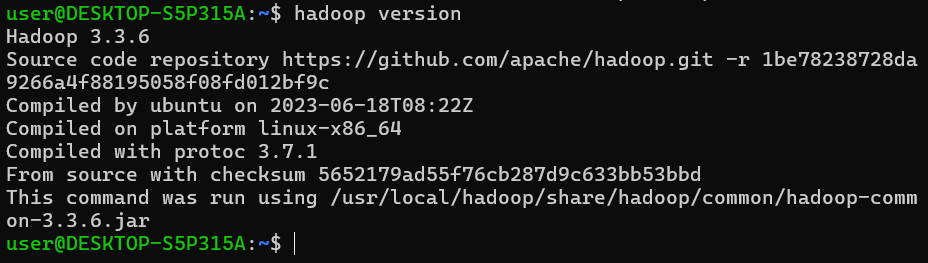
4. Download Java SE Development Kit



5. Configure hadoop



6. Check Hadoop version.



**Learning Outcome**

**Experiment 2**

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

**Theory**

This experiment focuses on developing a MapReduce program using Apache Hadoop to efficiently compute the frequency of a specified word within a given text file. MapReduce is a programming model designed for processing large datasets in a distributed manner. By utilizing Hadoop's parallel processing capabilities, this approach enables efficient text analysis and provides insights into word frequency.

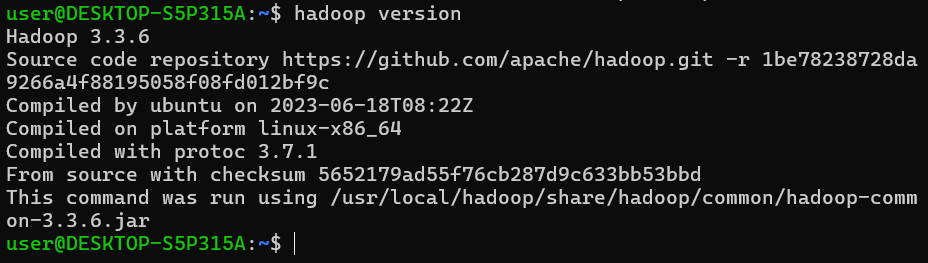
Word frequency analysis is a key task in natural language processing (NLP), involving the determination of how often each word appears in a document or corpus. Apache Hadoop offers a scalable framework for distributed computing, making it well-suited for parallel processing tasks like MapReduce. This experiment employs the MapReduce paradigm to distribute the workload across multiple nodes in a Hadoop cluster, significantly enhancing the speed and efficiency of text data analysis.

The primary objective of this experiment is to develop and implement a MapReduce program that calculates the frequency of a given word in a text file. By distributing computations across multiple nodes, this experiment highlights the scalability and efficiency of Hadoop in handling large-scale text processing tasks.

Ultimately, the experiment showcases the effectiveness of Apache Hadoop and the MapReduce framework in processing big data. By leveraging distributed computing, Hadoop enables efficient and scalable word frequency analysis, making it applicable to various domains requiring large-scale data processing.

**Steps**

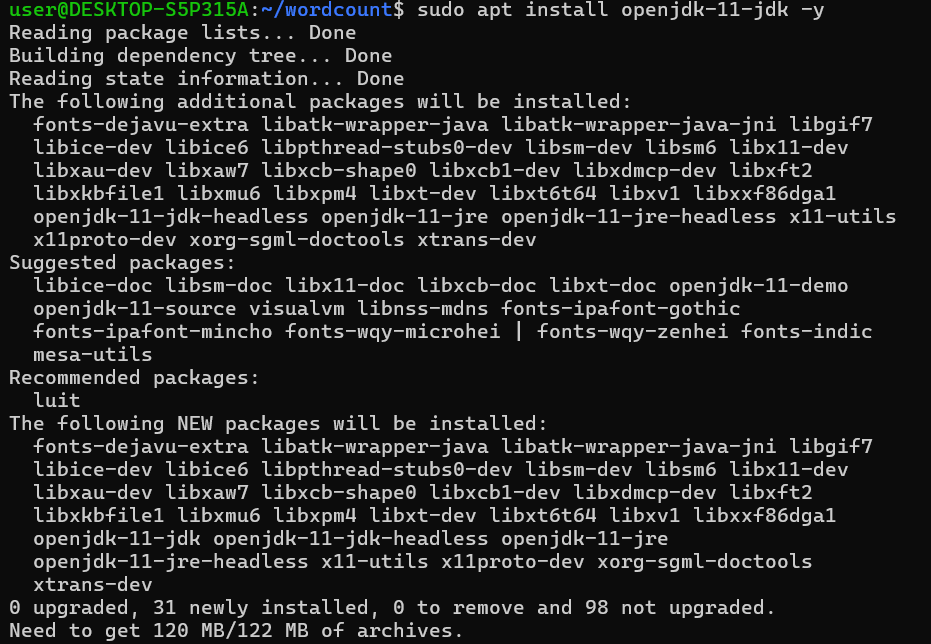
Check Hadoop version

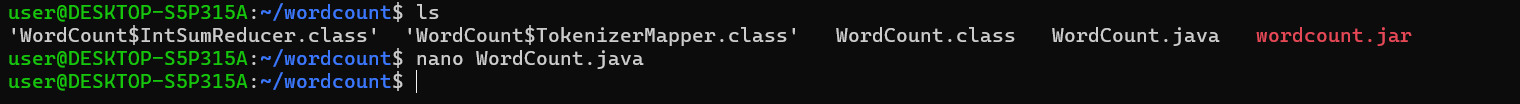


Create folder named wordcount



Install JDK



****

**Code of WordCount.java file**

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 java.io.IOException;

import java.util.StringTokenizer;

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);

}

}

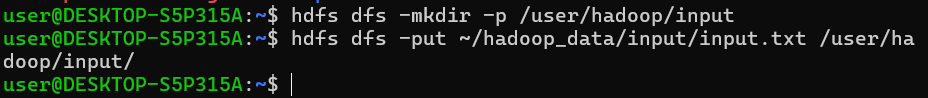
Compile java program

****

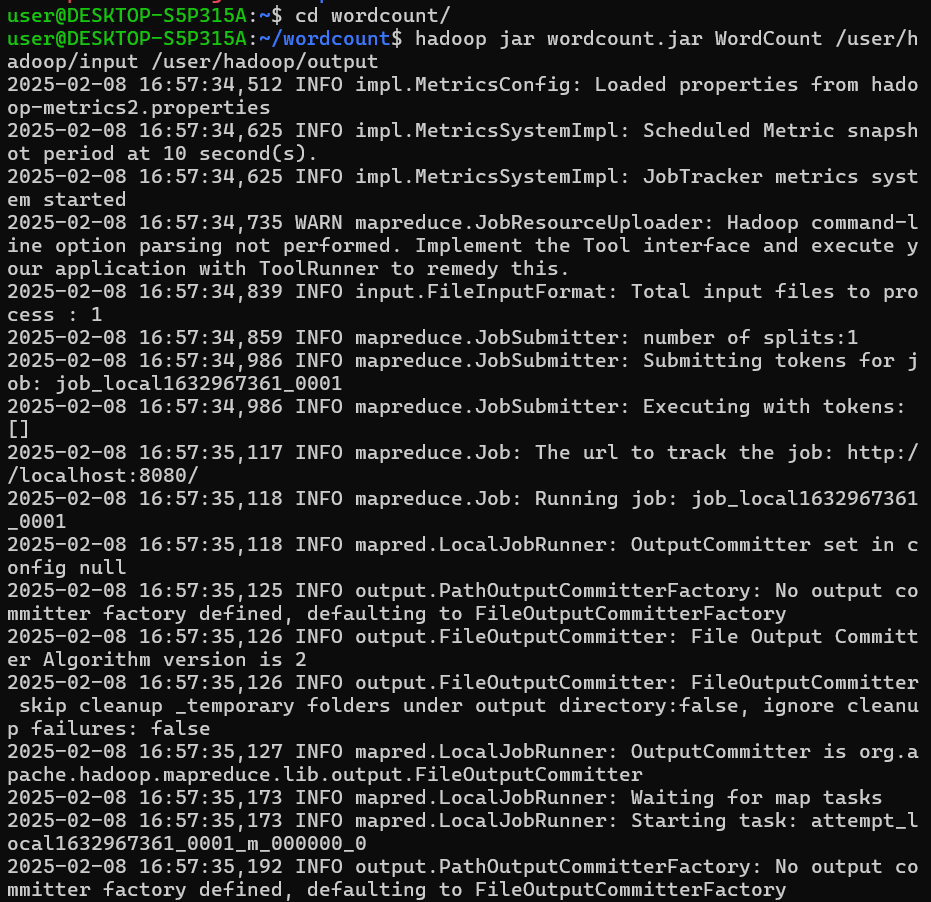
Create jar file

****

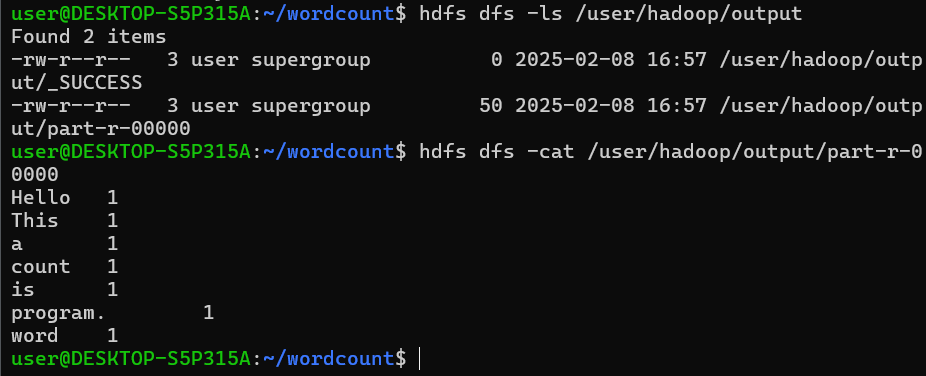
Copy the input file to Hadoop's HDFS:



Run the MapReduce Job



Check the output directory and Retrieve and display the word count:



**Learning Output**

**Experiment 3**

**Aim**: Develop a MapReduce program to find the grade of students.

**Theory:**

The aim of this MapReduce program is to process student scores and assign grades based on predefined score ranges. The program will take input in the form of student names and scores, then calculate each student's grade using a grading scale: A (90-100), B (80-89), C (70-79), D (60-69), and F (below 60). The MapReduce framework is ideal for this task because it can efficiently handle large datasets by distributing the processing across multiple machines, ensuring scalability and faster computation.

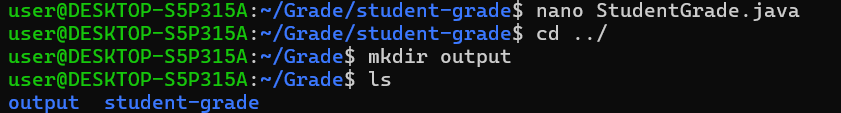
In the MapReduce process, the mapper will read each student’s score, determine the corresponding grade, and emit key-value pairs (student name, grade). The reducer will aggregate the results, ensuring each student's name is paired with their respective grade in the final output. This approach allows for parallel processing of large datasets, and the Hadoop framework ensures fault tolerance, making it a robust solution for determining grades across vast amounts of student data.

**Steps**

Create directories:



Create Java file:



**Java Code:**

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.Path;

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 java.io.IOException;

public class StudentGrade {

public static class GradeMapper extends Mapper<Object, Text, Text, Text> {

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

String[] parts = value.toString().split(",");

if (parts.length == 2) {

String studentName = parts[0].trim();

double score = Double.parseDouble(parts[1].trim());

String grade;

if (score >= 90) {

grade = "A";

} else if (score >= 80) {

grade = "B";

} else if (score >= 70) {

grade = "C";

} else if (score >= 60) {

grade = "D";

} else {

grade = "F";

}

context.write(new Text(studentName), new Text(grade));

}

}

}

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

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

for (Text val : values) {

context.write(key, val);

break;

}

}

}

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

if (args.length != 2) {

System.err.println("Usage: StudentGrade <input path> <output path>");

System.exit(-1);

}

Configuration conf = new Configuration();

Job job = Job.getInstance(conf, "Student Grade Calculation");

job.setJarByClass(StudentGrade.class);

job.setMapperClass(GradeMapper.class);

job.setReducerClass(GradeReducer.class);

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(Text.class);

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

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

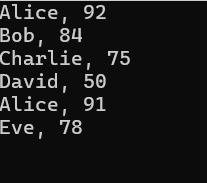
System.exit(job.waitForCompletion(true) ? 0 : 1);

}

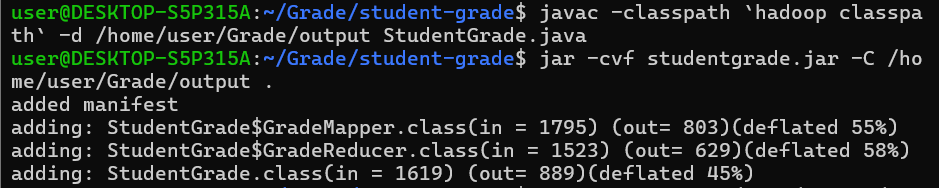
}

Create input file:





Compile java file and Jar file:



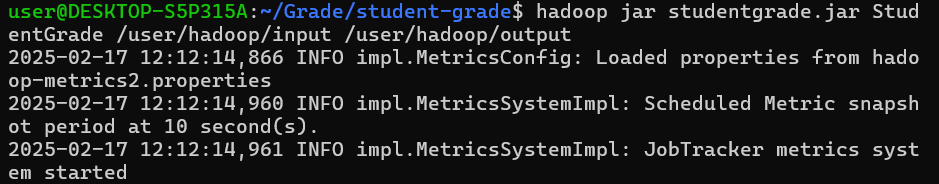
**Create the output directory** in HDFS (Hadoop File System):



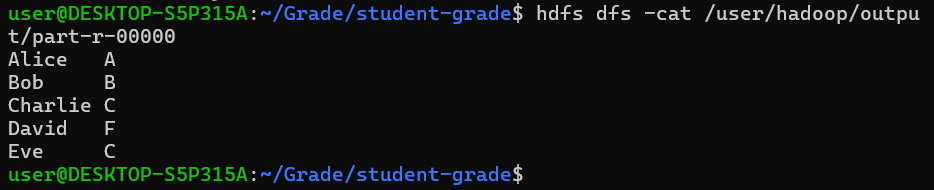
**Upload the input file to HDFS:**



**Run the MapReduce job** using the following command:



Check the Output:



**Learning Outcome**

**Experiment 4**

**Aim**: Develop a MapReduce program to find the maximum temperature in a year.

**Theory:**

The objective of this MapReduce program is to determine the maximum temperature recorded in a year from a large dataset containing weather records. In a Hadoop-based distributed computing environment, the **Map** function processes each line of input data, extracting the year and temperature values. It then emits key-value pairs where the key is the year, and the value is the temperature. The **Reduce** function receives all temperature values for each year, compares them, and outputs the highest temperature recorded for that year.

This approach is particularly useful for handling large-scale climate datasets efficiently. By leveraging Hadoop’s parallel processing capability, the program ensures that temperature data from multiple sources is processed simultaneously, making it faster and more scalable than traditional methods. The results can be used for climate analysis, weather forecasting, and environmental monitoring, helping researchers and meteorologists analyze temperature trends over time.

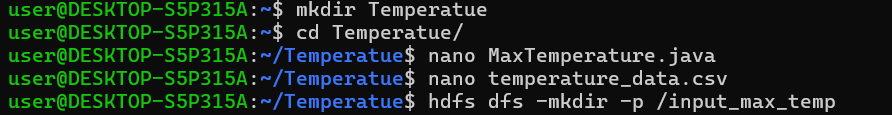
**Steps**

Create directories,

Create Java file,

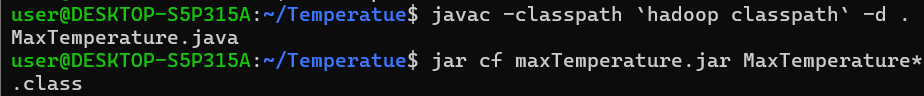
Create CSV file for temperature data

Create Directory in HDFS for input:



Compile Java File,

Create Jar file:



**Code:**

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.DoubleWritable;

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 java.io.IOException;

// Mapper Class

public class MaxTemperature {

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

private IntWritable year = new IntWritable();

private DoubleWritable temperature = new DoubleWritable();

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

String[] fields = value.toString().split(","); // Assuming CSV format

if (fields.length < 3) return; // Skip malformed lines

try {

year.set(Integer.parseInt(fields[0].trim())); // Extract Year

temperature.set(Double.parseDouble(fields[2].trim())); // Extract Temperature

context.write(year, temperature);

} catch (NumberFormatException e) {

// Skip invalid data

}

}

}

// Reducer Class

public static class TemperatureReducer extends Reducer<IntWritable, DoubleWritable, IntWritable, DoubleWritable> {

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

double maxTemp = Double.MIN\_VALUE;

for (DoubleWritable val : values) {

maxTemp = Math.max(maxTemp, val.get());

}

context.write(key, new DoubleWritable(maxTemp));

}

}

// Driver Code

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

Configuration conf = new Configuration();

Job job = Job.getInstance(conf, "Max Temperature Finder");

job.setJarByClass(MaxTemperature.class);

job.setMapperClass(TemperatureMapper.class);

job.setReducerClass(TemperatureReducer.class);

job.setMapOutputKeyClass(IntWritable.class);

job.setMapOutputValueClass(DoubleWritable.class);

job.setOutputKeyClass(IntWritable.class);

job.setOutputValueClass(DoubleWritable.class);

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

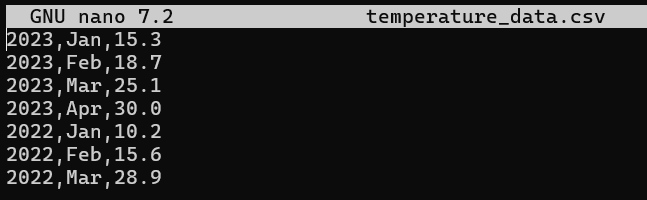
FileOutputFormat.setOutputPath(job, new Path(args[1])); // Output path

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

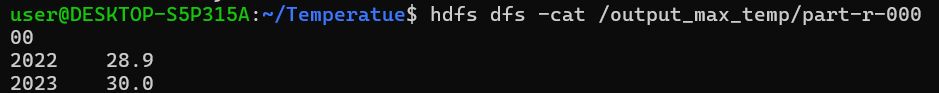
}

}

Input Temperature Data:



Output:



Learning Outcome:

**Experiment 5**

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

**Theory:**

Matrix multiplication using the MapReduce framework is a distributed approach to processing large matrices efficiently. The Map phase processes matrix elements independently, emitting key-value pairs where the key represents the target position in the result matrix. The Reduce phase then aggregates these intermediate values by performing the required multiplications and summations, producing the final matrix product. This approach enables scalable and parallel computation, making it suitable for handling massive datasets in distributed environments like Hadoop.

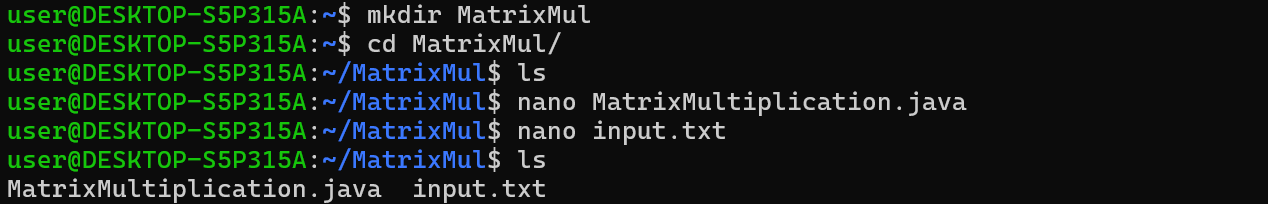
By leveraging the Hadoop Distributed File System (HDFS), matrix data is stored across multiple nodes, allowing for fault tolerance and efficient processing. The input matrices are read from HDFS, processed by the Mapper, and combined in the Reducer to generate the output matrix. This method is commonly used in scientific computing, machine learning, and big data analytics where matrix operations are fundamental.

**Steps**

Create directories,

Create Java file,

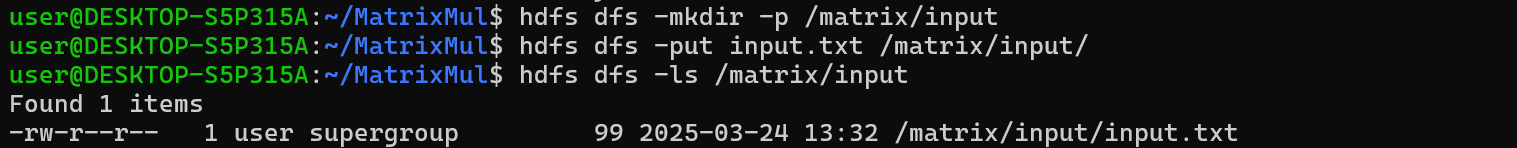
Create text file for two matrices:



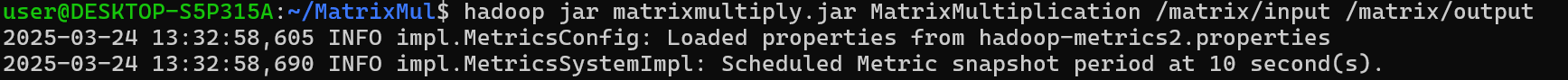
Compile java file and create jar file



Create Directory in HDFS for input:



Run MapReduce Job:



Code :

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.Path;

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 java.io.IOException;

import java.util.ArrayList;

import java.util.List;

public class MatrixMultiplication {

// Mapper Class

public static class MatrixMapper extends Mapper<Object, Text, Text, Text> {

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

String[] parts = value.toString().split("\\s+");

if (parts.length < 4) return;

String matrixName = parts[0]; // 'A' or 'B'

int i = Integer.parseInt(parts[1]);

int j = Integer.parseInt(parts[2]);

double val = Double.parseDouble(parts[3]);

Configuration conf = context.getConfiguration();

int p = Integer.parseInt(conf.get("p")); // Columns of B

if (matrixName.equals("A")) {

for (int k = 0; k < p; k++) {

context.write(new Text(i + "," + k), new Text("A," + j + "," + val));

}

} else if (matrixName.equals("B")) {

int m = Integer.parseInt(conf.get("m")); // Rows of A

for (int k = 0; k < m; k++) {

context.write(new Text(k + "," + j), new Text("B," + i + "," + val));

}

}

}

}

// Reducer Class

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

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

List<String> aValues = new ArrayList<>();

List<String> bValues = new ArrayList<>();

for (Text val : values) {

String valueStr = val.toString();

if (valueStr.startsWith("A")) {

aValues.add(valueStr);

} else {

bValues.add(valueStr);

}

}

double sum = 0;

for (String a : aValues) {

String[] aParts = a.split(",");

int aCol = Integer.parseInt(aParts[1]);

double aVal = Double.parseDouble(aParts[2]);

for (String b : bValues) {

String[] bParts = b.split(",");

int bRow = Integer.parseInt(bParts[1]);

double bVal = Double.parseDouble(bParts[2]);

if (aCol == bRow) {

sum += aVal \* bVal;

}

}

}

context.write(key, new Text(Double.toString(sum)));

}

}

// Main Method

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

Configuration conf = new Configuration();

conf.set("m", "2"); // Rows of A

conf.set("n", "3"); // Columns of A / Rows of B

conf.set("p", "2"); // Columns of B

Job job = Job.getInstance(conf, "Matrix Multiplication");

job.setJarByClass(MatrixMultiplication.class);

job.setMapperClass(MatrixMapper.class);

job.setReducerClass(MatrixReducer.class);

job.setMapOutputKeyClass(Text.class);

job.setMapOutputValueClass(Text.class);

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(Text.class);

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

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

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

}

}

Input Text File:

A 0 0 1

A 0 1 2

A 0 2 3

A 1 0 4

A 1 1 5

A 1 2 6

B 0 0 7

B 1 0 8

B 2 0 9

B 0 1 10

B 1 1 11

B 2 1 12

Show Result:



**Learning Outcome:**

**Experiment 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:**

MapReduce is a distributed computing framework that enables parallel processing of large datasets. In this program, we use MapReduce to determine the maximum monthly electricity consumption for each year from a dataset containing yearly, monthly, and consumption values. The Mapper extracts the year and consumption values, emitting key-value pairs where the key is the year and the value is the consumption. The Reducer then processes all values corresponding to each year and finds the maximum consumption for that year.

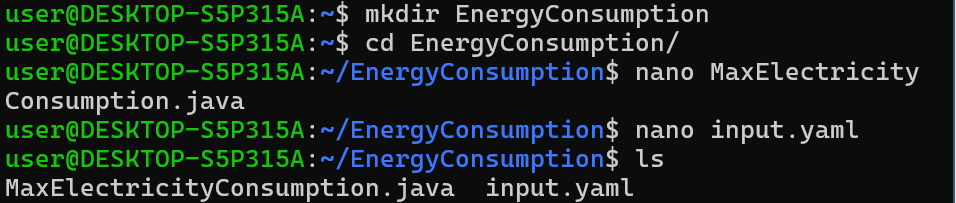
Hadoop Distributed File System (HDFS) stores the dataset across multiple nodes, allowing the MapReduce job to process it efficiently. This approach is particularly useful in big data analytics, where large-scale computations need to be performed on distributed clusters. By utilizing parallelism, MapReduce improves performance and scalability, making it an ideal choice for handling large datasets such as electricity consumption records over multiple years.

**Steps**

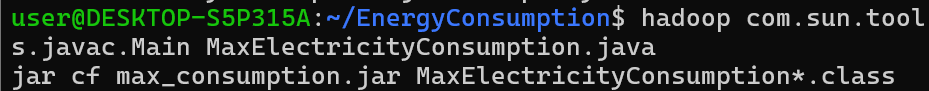
Create directories,

Create Java file,

Create text file for two matrices:



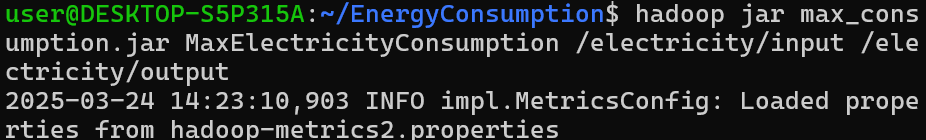
Compile java file and Create Jar file



Create an HDFS directory and upload input data:



Run MapReduce Job:



Code:

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 java.io.IOException;

public class MaxElectricityConsumption {

// Mapper Class

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

private Text year = new Text();

private IntWritable consumption = new IntWritable();

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

String[] fields = value.toString().split(",");

if (fields.length == 3) {

year.set(fields[0]); // Extract Year

consumption.set(Integer.parseInt(fields[2])); // Extract Consumption

context.write(year, consumption);

}

}

}

// Reducer Class

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

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

int maxConsumption = Integer.MIN\_VALUE;

for (IntWritable val : values) {

maxConsumption = Math.max(maxConsumption, val.get());

}

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

}

}

// Driver Code

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

Configuration conf = new Configuration();

Job job = Job.getInstance(conf, "Max Electricity Consumption");

job.setJarByClass(MaxElectricityConsumption.class);

job.setMapperClass(MaxConsumptionMapper.class);

job.setReducerClass(MaxConsumptionReducer.class);

job.setMapOutputKeyClass(Text.class);

job.setMapOutputValueClass(IntWritable.class);

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(IntWritable.class);

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

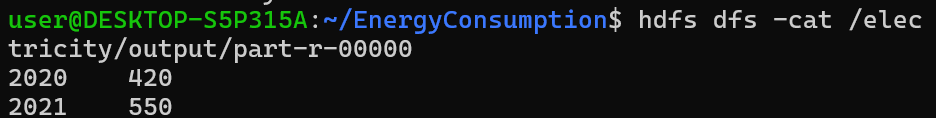
FileOutputFormat.setOutputPath(job, new Path(args[1]));

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

}

}

Output:



Learning Outcome:

**Experiment 7**

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

**Theory:**

To analyze a weather dataset using Hadoop, a MapReduce program is developed which processes large volumes of weather data in a distributed manner. The process starts by placing the input dataset into the Hadoop Distributed File System (HDFS), which splits the data into blocks and stores them across multiple nodes. The Map function reads the input line by line, excluding the header, and parses each record to extract relevant fields like date, temperature, and weather condition. Based on defined criteria (e.g., temperature > 25°C and condition is "Sunny" for a Shiny day, otherwise a Cool day), it emits a key-value pair with the date as the key and day type as the value.

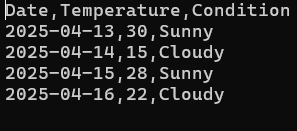
The Reduce function simply collects these key-value pairs and writes the final output, which indicates whether each day in the dataset is shiny or cool. The technology stack used includes Hadoop MapReduce for distributed data processing, HDFS for storage, and Java as the programming language. This setup showcases the efficiency of Hadoop in handling large datasets and performing parallel processing to derive meaningful insights such as weather categorization.

**Steps:**

**Create text and java file**

****

**Input**

****

**Java File**

import java.io.IOException;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.DoubleWritable;

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 WeatherMapReduce {

public static class WeatherMapper extends Mapper<Object, Text, Text, DoubleWritable> {

@Override

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

String line = value.toString();

// Skip header line

if (line.startsWith("Date")) return;

try {

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

if (parts.length >= 2) {

String date = parts[0].trim();

double temperature = Double.parseDouble(parts[1].trim());

context.write(new Text(date), new DoubleWritable(temperature));

}

} catch (NumberFormatException e) {

System.err.println("Error parsing line: " + line);

}

}

}

public static class WeatherReducer extends Reducer<Text, DoubleWritable, Text, DoubleWritable> {

@Override

public void reduce(Text key, Iterable<DoubleWritable> values, Context context)

throws IOException, InterruptedException {

double sum = 0;

int count = 0;

for (DoubleWritable val : values) {

sum += val.get();

count++;

}

if (count > 0) {

double average = sum / count;

context.write(key, new DoubleWritable(average));

}

}

}

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

Configuration conf = new Configuration();

Job job = Job.getInstance(conf, "Average Temperature");

job.setJarByClass(WeatherMapReduce.class);

job.setMapperClass(WeatherMapper.class);

job.setReducerClass(WeatherReducer.class);

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(DoubleWritable.class);

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

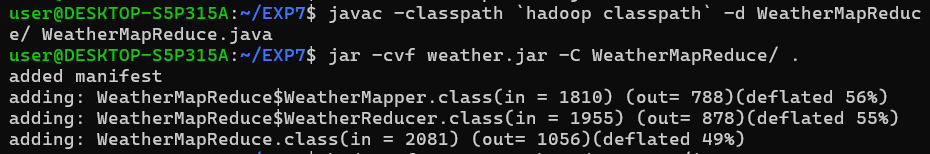
FileOutputFormat.setOutputPath(job, new Path(args[1]));

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

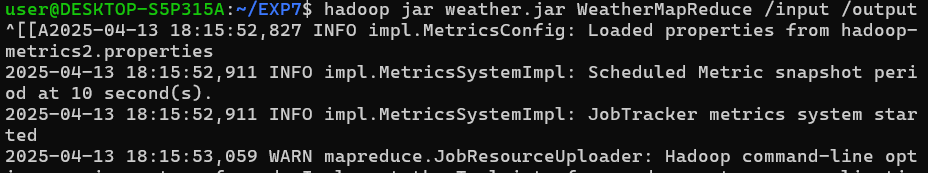
}

}

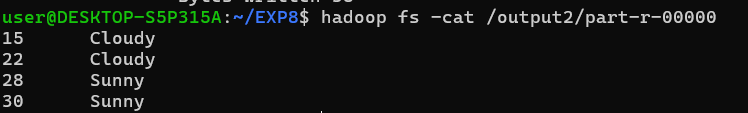
**Compile Java file and create Jar file**

****

**Upload Input dataset in HDFS**

****

**Run output file**

****

**Learning Outcome**

**Experiment 8**

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

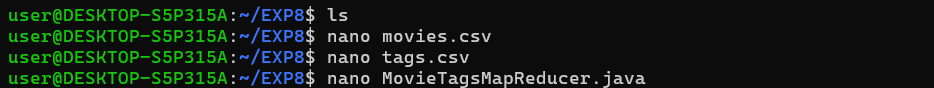
**Theory:**

The MapReduce program is designed to analyze the MovieLens dataset and find the tags associated with each movie. The process involves two main components: the Mapper and the Reducer. The Mapper processes each line of input data, extracting the movie ID and associated tag, and then emits key-value pairs with the movie ID as the key and the tag as the value. This allows the Mapper to organize the data in a way that can be easily aggregated in the next step. In this case, the Mapper will read lines of data such as 1, Cloudy and emit (1, Cloudy).

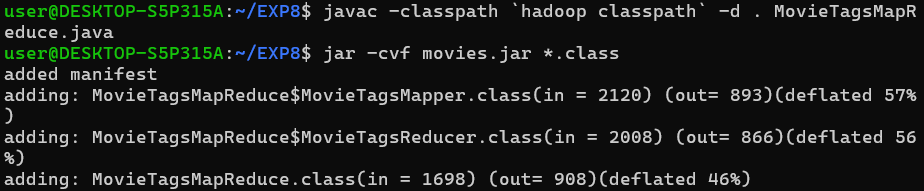
The Reducer takes the intermediate key-value pairs from the Mapper, groups the tags by movie ID, and aggregates them to form a list of tags for each movie. The Reducer’s role is to ensure that all tags related to a movie are combined into a single output, allowing the system to produce the final result, which is a list of tags for each movie. This MapReduce framework efficiently processes large datasets by distributing the work across multiple nodes, enabling parallel processing and reducing the time required for large-scale data analysis.

**Steps**

**Create Files Movies.csv, tags.csv and MovieTagsMapReducer.java**

****

**Compile java file and create jar file**

****

**Java Code:**

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.Path;

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

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

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

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

import java.io.IOException;

import java.util.StringTokenizer;

public class MovieTagsMapReduce {

// Mapper Class

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

private final static IntWritable movieId = new IntWritable();

private final static Text tag = new Text();

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

// Skip the header

if (key.get() == 0) return;

// Split the input line by commas (assuming tags.csv format: userId, movieId, tag, timestamp)

String line = value.toString();

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

try {

// Parse movieId and tag

int movieIdInt = Integer.parseInt(fields[1]);

String movieTag = fields[2];

// Set movieId and tag to be emitted as output

movieId.set(movieIdInt);

tag.set(movieTag);

// Write the movieId as key and tag as value

context.write(movieId, tag);

} catch (NumberFormatException e) {

// Skip rows where movieId is not an integer

}

}

}

// Reducer Class

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

private Text result = new Text();

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

// StringBuilder to collect all tags for a particular movieId

StringBuilder tagsList = new StringBuilder();

for (Text val : values) {

// Append each tag to the list, separating with commas

if (tagsList.length() > 0) {

tagsList.append(", ");

}

tagsList.append(val.toString());

}

// Set the final result in the form of "movieId: tag1, tag2, tag3..."

result.set(tagsList.toString());

context.write(key, result);

}

}

// Main Driver Method

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

// Check for correct number of arguments

if (args.length != 2) {

System.err.println("Usage: MovieTagsMapReduce <input path> <output path>");

System.exit(-1);

}

// Configuration setup

Configuration conf = new Configuration();

Job job = Job.getInstance(conf, "Movie Tags MapReduce");

job.setJarByClass(MovieTagsMapReduce.class);

job.setMapperClass(MovieTagsMapper.class);

job.setReducerClass(MovieTagsReducer.class);

job.setOutputKeyClass(IntWritable.class);

job.setOutputValueClass(Text.class);

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

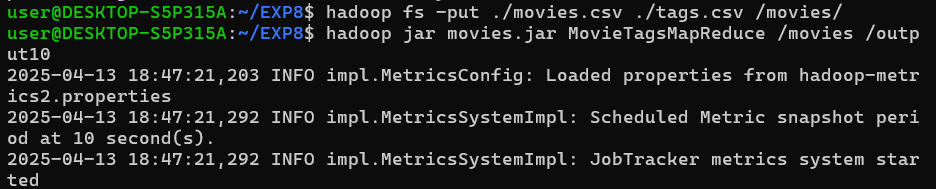
FileOutputFormat.setOutputPath(job, new Path(args[1]));

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

}

}

**Put files in HDFS run jar**

****

**Output**

****

**Learning Outcome**

**Experiment 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 they are:

Dispatching, base, no. date active, vehicle trips.

**Theory:**

This practical experiment involves analyzing Uber trip data using the Hadoop MapReduce framework, a powerful tool for processing large datasets in a distributed computing environment. The dataset contains records of Uber trips, including fields such as dispatching base, date, and the number of trips. The objective is to compute the total number of trips taken for each combination of base and date. The MapReduce model follows a divide-and-conquer approach where the Mapper extracts key-value pairs from each line of the dataset (with the key as "Base \t Date" and the value as the number of trips), and the Reducer aggregates these values to compute the total trips for each unique key.

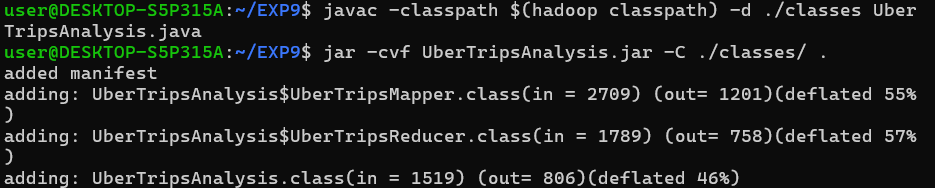
The program handles data preprocessing, such as skipping header lines and filtering malformed records, ensuring only valid data contributes to the final output. The result is a cleaned and summarized dataset that provides insights into the distribution and volume of Uber trips over time and across dispatch bases. This experiment demonstrates the core concepts of Hadoop MapReduce: mapping, shuffling, and reducing, and how they can be used for efficient and scalable big data analysis in real-world transportation datasets.

**Steps**

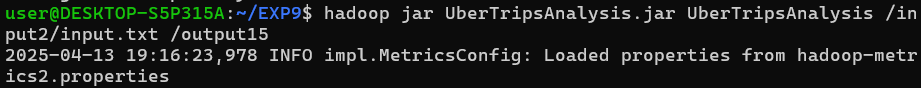
**Create Files input.txt and UberTripsAnalysis.java**

****

**Compile java file and create jar file**

****

**Upload files to HDFS**

****

**Java Program:**

import java.io.IOException;

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 UberTripsAnalysis {

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

private final static IntWritable tripCount = new IntWritable(0);

private Text baseAndDate = new Text();

private boolean isHeader(String line) {

return line.startsWith("Dispatching") || line.startsWith("Base");

}

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

String line = value.toString().trim();

if (line.isEmpty() || isHeader(line)) {

return; // Skip headers or empty lines

}

String[] fields = line.split(",", -1); // -1 to include trailing empty fields

if (fields.length != 4) {

return; // Skip invalid lines

}

try {

String base = fields[1].trim();

String date = fields[2].trim();

int trips = Integer.parseInt(fields[3].trim());

baseAndDate.set(base + "\t" + date);

tripCount.set(trips);

context.write(baseAndDate, tripCount);

} catch (NumberFormatException e) {

// Ignore lines with invalid trip count

}

}

}

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

private IntWritable totalTrips = 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();

}

totalTrips.set(sum);

context.write(key, totalTrips);

}

}

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

Configuration conf = new Configuration();

Job job = Job.getInstance(conf, "Uber Trips Analysis");

job.setJarByClass(UberTripsAnalysis.class);

job.setMapperClass(UberTripsMapper.class);

job.setReducerClass(UberTripsReducer.class);

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(IntWritable.class);

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

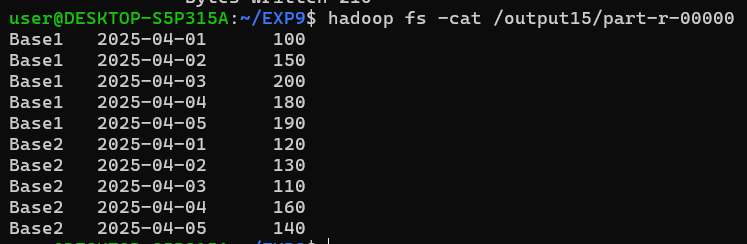
FileOutputFormat.setOutputPath(job, new Path(args[1]));

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

}

}

**Output**

****

**Leaning Outcome**