EXPERIMENT 1

Introduction to Ubuntu
Ubuntu is an open-source Linux distribution based on Debian, widely used for development and server environments. It is known for its stability, security, and ease of use. Ubuntu provides a command-line interface (CLI) and graphical user interface (GUI), making it suitable for both beginners and advanced users. It is often used for cloud computing, artificial intelligence, and big data applications.

Apache Hadoop

Apache Hadoop is an open-source framework designed for distributed storage and processing of large datasets. It follows a master-slave architecture, where the master node (NameNode) manages metadata, and slave nodes (DataNodes) store actual data. Hadoop is built to handle petabytes of data efficiently using the Hadoop Distributed File System (HDFS) and the MapReduce programming model.

Hadoop is highly scalable, allowing users to add more nodes as data volume increases. It supports multiple processing engines such as Apache Spark, Hive, and Pig, enabling a wide range of data analytics and machine learning applications. Many cloud providers, including AWS and Google Cloud, offer Hadoop-based services, making it accessible for large-scale data processing.

Hadoop Framework and Installation Steps

Hadoop consists of four main components:

1. **HDFS (Hadoop Distributed File System)** – Manages large data storage across multiple nodes.
2. **MapReduce** – Handles parallel processing of data.
3. **YARN (Yet Another Resource Negotiator)** – Manages resources for processing tasks.
4. **Hadoop Common** – Provides essential libraries and utilities for Hadoop operations.

To install Hadoop, Ubuntu is used as the operating system. The installation process involves downloading the Hadoop binary package, extracting it, and configuring environment variables such as `HADOOP_HOME` and `JAVA_HOME`. Java is required for Hadoop to function, so a Java Runtime Environment (JRE) must be installed.

Once the setup is complete, commands like `hadoop version` verify the installation. Hadoop can then be configured for single-node or multi-node clusters based on the use case.

- Open Oracle VirtualBox.
- Start the Ubuntu virtual machine.
- Wait for the Ubuntu home screen after logging in.
- Open the Apache Hadoop official website in a web browser.
- Navigate to the Hadoop download page.
- Select the Hadoop 3.4.1 binary version and copy the download URL.
- Open the terminal in Ubuntu.

- Run `wget https://dlcdn.apache.org/hadoop/common/hadoop-3.4.1/hadoop-3.4.1.tar.gz` to download Hadoop.
- Extract the downloaded file using `tar xzf hadoop-3.4.1.tar.gz`.
- Install Java Runtime Environment with `sudo apt install default-jre`.
- Open the profile configuration file using `sudo gedit /etc/profile`.
- Insert the following lines in `/etc/profile`:
- Apply the changes using `source /etc/profile`.
- Verify Java installation with `echo $JAVA_HOME` and `java -version`.
- Check the Hadoop environment variable using `echo $HADOOP_HOME`.
- Confirm the installation by running `hadoop version`.

Understanding the installation and configuration of Apache Hadoop on Ubuntu, including setting up environment variables and verifying system compatibility for distributed data processing.

Understanding the verification process using commands like `echo $JAVA_HOME`, `echo $HADOOP_HOME`, and `hadoop version` to ensure proper installation and system recognition.

EXPERIMENT 2

Understanding MapReduce

MapReduce is a distributed computing framework used for processing large datasets in parallel across multiple nodes in a cluster. It is a fundamental part of Apache Hadoop, allowing efficient handling of massive amou nts of data. The framework divides tasks into two main functions: the **Map function**, which processes input data and generates intermediate key-value pairs, and the **Reduce function**, which aggregates these key-value pairs to produce the final output. This model is fault-tolerant and optimized for distributed environments, making it ideal for big data processing. MapReduce is commonly used for log analysis, text processing, and large-scale computations where traditional methods are inefficient.

Functioning of MapReduce

MapReduce follows a systematic workflow, beginning with input data stored in Hadoop Distributed File System (HDFS). The input is divided into chunks, and each chunk is processed independently by the Mapper function. The Mapper converts raw data into key-value pairs, such as words in a document mapped to the number 1. The intermediate results are then shuffled and sorted, grouping similar keys together. Once grouped, the Reducer function processes these keys, aggregating the values to compute the final results. For example, in a word count program, the Reducer sums all occurrences of each word, producing the total frequency. The output is then written back to HDFS for storage and further analysis. This process ensures efficient utilization of computational resources while handling vast amounts of data.

Implementing Word Frequency Calculation Using MapReduce

The experiment involves implementing a **Word Count** program using MapReduce to determine the frequency of words in a given file. A Java program, WordCount.java, is created using the **nano** editor,

containing the logic for the Mapper and Reducer functions. The program is then compiled using the javac command, generating necessary class files for execution. A JAR file is created using the jar command to package the compiled files for Hadoop execution. A sample text file is generated, containing words to be processed. The Hadoop job is executed using the command hadoop jar wc.jar WordCount sample.txt output, where Hadoop distributes the file across nodes and applies the MapReduce logic. Once the job is completed successfully, the output is stored in HDFS. The results are viewed using the command hdfs dfs -cat output/part-r-00000, displaying word frequencies in a key-value format.

- Create `WordCount.java` using the `nano` text editor in the terminal.
- Enter the MapReduce program code for word frequency calculation, then save and exit.
- Compile `WordCount.java` using `javac` to generate the required class files.
- Package the compiled class files into a JAR file using the `jar` command.
- Create a sample text file using `nano` to provide input data for the Hadoop job.
- Enter sample text into the file, then save and exit.
- Run the Hadoop job using `hadoop jar wc.jar WordCount sample.txt output` to process the text file.
- Wait for the Hadoop job to complete successfully, ensuring no errors occur.
- View the processed output using `hdfs dfs -cat output/part-r-00000` to check word frequencies.

 Understanding the MapReduce framework and its role in distributed data processing by implementing a word frequency program using Hadoop, ensuring efficient parallel computation.

 Gaining hands-on experience in writing, compiling, packaging, and executing a Java-based MapReduce program while utilizing HDFS for storing and retrieving processed data.

EXPERIMENT 3

MapReduce Framework

MapReduce is a distributed computing model used in Hadoop to process large-scale datasets efficiently. It follows a **divide and conquer** approach, where tasks are split into smaller parts and processed in parallel across multiple nodes. The framework consists of two main phases:

- **Mapper Phase**: Processes input data and emits key-value pairs.
- **Shuffle and Sort**: Groups values by key and sorts them before sending them to the reducer.
- **Reducer Phase**: Aggregates the values for each key to produce the final result.

This model is widely used for data-intensive computations like log analysis, word counting, and temperature calculations.

Finding Maximum Temperature using MapReduce

In this experiment, MapReduce is applied to determine the maximum temperature recorded in each year from a dataset. The dataset typically contains multiple records in the format:

Year Temperature
2010 32
2010 35
2011 30
2011 40

Mapper Function

The mapper reads each line of the dataset, extracts the year and temperature, and emits key-value pairs in the format **(year, temperature)**. These pairs serve as input for further processing.

Shuffle and Sort Phase

This phase groups all temperatures corresponding to the same year and sorts them if needed. Hadoop ensures that values for each key are correctly aggregated before reaching the reducer.

Reducer Function

The reducer receives the grouped temperature values for each year, scans through them, and determines the highest recorded temperature. It then emits the final result as **(year, max temperature)**.

Execution in Hadoop

The dataset is stored in **HDFS** for distributed processing. The Hadoop job runs using `hadoop jar`, ensuring efficient computation across multiple nodes. The final output, containing the maximum temperature for each year, is stored in an HDFS output directory.

By leveraging MapReduce, the process efficiently handles large-scale weather datasets, making temperature analysis scalable and reliable.

- Create `Max_temp.java` using the `nano` text editor in the terminal.
- Enter the MapReduce program code to find maximum temperature, then save and exit.
- Compile `Max_temp.java` using `javac` to generate the required class files.
- Package the compiled class files into a JAR file using the `jar` command.
- Create a temperature.txt file using `nano` to provide input data for the Hadoop job.
- Enter year-temperature data into the file, then save and exit.
- Run the Hadoop job using `hadoop jar mt.jar Max_temp Temperature.txt output_2` to process the text file.
- Wait for the Hadoop job to complete successfully, ensuring no errors occur.
- View the processed output using `hdfs dfs -cat output_2/part-r-00000` to check word frequencies.

 Understanding the MapReduce framework to process large datasets by implementing a program that extracts and calculates the maximum temperature for each year.

 Gaining hands-on experience with Hadoop commands, including storing data in HDFS, running MapReduce jobs, and retrieving results efficiently.

EXPERIMENT 4

MapReduce for Student Grade Calculation

MapReduce is a powerful framework used for parallel processing of large datasets. It follows a **divide and conquer** approach, where tasks are divided into smaller parts and executed across multiple nodes. In this experiment, MapReduce is used to determine student grades based on their scores.

Processing Student Data with MapReduce

The dataset typically contains student records in the format:
Alice, 85
Bob, 73

Each student's marks are processed using **Mapper**, **Shuffle and Sort**, and **Reducer** phases.

Mapper Function

The mapper reads student records line by line from the dataset. It extracts relevant information, including the student ID, name, and marks. The mapper then emits key-value pairs in the format **(StudentName, Marks)**, which will be processed further in the next stages.

Shuffle and Sort Phase

In this phase, Hadoop groups student records based on **StudentName** to ensure that all marks belonging to the same student are processed together. Sorting may also be performed if needed to organize the data efficiently for the reducer.

Reducer Function

The reducer receives the marks for each student and assigns a grade based on a predefined grading scale:

- **90–100 → A**
- **80–89 → B**
- **70–79 → C**
- **60–69 → D**
- **Below 60 → F**

Once the grades are assigned, the reducer emits the final output in the format **(StudentName, Grade)**.

Execution in Hadoop

The student data is stored in **HDFS** to enable distributed processing. The Hadoop job runs using the `hadoop jar` command, efficiently computing results across multiple nodes. After execution, the final output, containing student grades, is stored in an HDFS directory.

By leveraging MapReduce, the grading process is automated and scalable, making it an effective solution for handling large student datasets.

- Create `StudentGrades.java` using the `nano` text editor in the terminal.
- Enter the MapReduce program code to find grade of students, then save and exit.
- Compile `StudentGrades.java` using `javac` to generate the required class files.
- Package the compiled class files into a JAR file using the `jar` command.
- Create a Gardes.txt file using `nano` to provide input data for the Hadoop job.
- Enter StudentName-Marks data into the file, then save and exit.
- Run the Hadoop job using `hadoop jar sg.jar StudentGrades Grades.txt output_3` to process the text file.
- Wait for the Hadoop job to complete successfully, ensuring no errors occur.
- View the processed output using `hdfs dfs -cat output_3/part-r-00000` to check word frequencies

 Understanding how the MapReduce framework processes student records by reading data, extracting marks, and names, and emitting key-value pairs for further processing. Learning how the shuffle and sort phase groups records efficiently for accurate grading.

 Gaining practical experience in running Hadoop jobs, storing large datasets in HDFS, and automating the grading process using reducers and how distributed computing enhances performance and scalability for handling massive student datasets.

EXPERIMENT 5

MapReduce Framework for Matrix Multiplication

MapReduce is a powerful framework used for parallel and distributed data processing. It divides large computations into smaller tasks, which are processed independently across multiple nodes. The framework consists of:

- **Mapper Phase**: Reads input data, processes it, and emits key-value pairs.
- **Shuffle and Sort**: Groups values with the same key for aggregation.
- **Reducer Phase**: Performs computations on grouped data and produces the final output.

Matrix multiplication is a common computational task where two matrices are combined to form a third matrix. Each value in the result matrix depends on corresponding values from the two input matrices. Given two matrices **A** and **B**, their multiplication results in a new matrix **C**, where each value in **C** is derived from multiple values in **A** and **B**.

Mapper Function

The mapper reads matrix elements in the format **(MatrixName, Row, Column, Value)** and emits key-value pairs as **(i, k)** for matrix **A** and **(k, j)** for matrix **B**. This ensures that elements contributing to the same position in **C** are grouped together, allowing efficient distributed computation.

Shuffle and Sort Phase

Hadoop groups matrix elements based on common keys, ensuring that all values needed for computing **C(i, j)** are processed together. The shuffle phase directs elements of **A** and **B** for the same computation to the same reducer.

Reducer Function

The reducer receives grouped values for **(i, j)**, iterates through **A** and **B**, performs element-wise multiplication, and sums the results. It then emits the final computed value for **C(i, j)**.

Execution in Hadoop

Matrices are stored in **HDFS**, and the job runs using hadoop jar, distributing computations across nodes. The final matrix product is written to an **HDFS** output directory, leveraging Hadoop's efficiency for large-scale multiplication.

- Create `MatrixMultiplication.java` using the `nano` text editor in the terminal.
- Enter the MapReduce program code for matrix multiplication, then save and exit.
- Compile `MatrixMultiplication.java` using `javac` to generate the required class files.
- Package the compiled class files into a JAR file using the `jar` command.
- Create a matrix.txt file using `nano` to provide input data for the Hadoop job.
- Enter MatrixName-Row-Column-Value data into the file, then save and exit.
- Run the Hadoop job using `hadoop jar mm.jar MatrixMultiplication matrix.txt output_1` to process the text file.
- Wait for the Hadoop job to complete successfully, ensuring no errors occur.
- View the processed output using `hdfs dfs -cat output_1/part-r-00000` to check the calculated matrix

 Gaining an understanding of how the MapReduce framework can be applied to matrix multiplication by distributing the computation across multiple nodes for efficiency.

 Learning how to use Hadoop to process large-scale data in parallel, with elements grouped and multiplied in the reducer to obtain the final matrix product.

EXPERIMENT 6

MapReduce enables large-scale data processing by distributing computations across multiple nodes. It consists of a **Mapper** that extracts relevant data and a **Reducer** that aggregates results. This framework efficiently handles massive datasets, making it ideal for analyzing trends in electrical consumption over time.

In this experiment, we analyze monthly electrical consumption data to determine the highest consumption for each year. The dataset contains records in the format (Year, Month, Consumption).

Mapper Function:

- Reads input records containing year, month, and electricity consumption.
- Extracts the year and corresponding consumption value.

- Emits key-value pairs as (year, consumption) for further aggregation.

Shuffle and Sort Phase:

- Groups consumption values based on the year to ensure they are processed together.
- Sorts records to optimize the reduction phase.

Reducer Function:

- Receives all consumption values for a given year.
- Iterates through the values to identify the maximum consumption.
- Emits (year, max consumption) as the final output.

Execution in Hadoop:

- Data is stored in HDFS, enabling parallel and efficient processing.
- The MapReduce job is executed using hadoop jar, distributing computation across nodes.
- The output, containing yearly maximum consumption, is stored in HDFS.

This method ensures efficient analysis of large-scale electrical consumption data, providing valuable insights into peak usage trends over different years.

 Create MaxElectricConsumption.java using the nano text editor in the terminal.
 Enter the MapReduce program code to find max electrical consumption, then save and exit.
 Compile MaxElectricConsumption.java using javac to generate the required class files.
 Package the compiled class files into a JAR file using the jar command.
 Create a input.txt file using nano to provide input data for the Hadoop job.
 Enter Year-month-consumption data into the file, then save and exit.
 Run the Hadoop job using hadoop jar mec.jar MaxElectricConsumption input.txt output_6 to process the text file.
 Wait for the Hadoop job to complete successfully, ensuring no errors occur.
 View the processed output using hdfs dfs -cat output_6/part-r-00000 to check the max electric consumption per year

 Gained an understanding of how MapReduce processes large datasets efficiently by extracting, grouping, and aggregating data.
 Learned to implement MapReduce for analyzing yearly electrical consumption trends, optimizing data processing in Hadoop.

EXPERIMENT 7

MapReduce is a **distributed computing framework** used for processing large datasets efficiently. It follows a **divide-and-conquer** approach, splitting tasks into **Mapper and Reducer** phases. The **Mapper** processes input data and emits key-value pairs, while the **Reducer** aggregates and processes these values. Hadoop's **HDFS** enables parallel execution, making it ideal for big data tasks.

Weather Analysis using MapReduce:

- The dataset consists of **dates and temperature values** as input.
- The **Mapper** extracts each record, using the **date as the key** and **temperature as the value**.
- The **Shuffle and Sort** phase groups temperatures by date for further processing.
- The **Reducer** classifies each day as **Shiny Day** if the temperature is above **30°C**, otherwise, it is a **Cool Day**.

Execution in Hadoop:

- **HDFS** stores the weather dataset for distributed access.
- The **MapReduce job** runs in parallel across multiple nodes for faster computation.
- The output, which contains classified weather conditions, is stored in **HDFS**.

Why Use MapReduce for Weather Analysis?

- **Scalability** – Handles large weather datasets efficiently.
- **Parallel Processing** – Reduces computation time by distributing tasks.
- **Automation** – Eliminates manual classification of weather conditions.

This method provides an efficient way to analyze weather trends and categorize days based on temperature variations using **Hadoop's distributed computing power**.

 Create WeatherAnalysis.java using the nano text editor in the terminal.
 Enter the MapReduce program code to find analysed weather, then save and exit.
 Compile WeatherAnalysis.java using javac to generate the required class files.
 Package the compiled class files into a JAR file using the jar command.
 Create a weatherdata.txt file using nano to provide input data for the Hadoop job.
 Enter date-value data into the file, then save and exit.
 Run the Hadoop job using hadoop jar wa.jar WeatherAnalysis weatherdata.txt outputwa to process the text file.
 Wait for the Hadoop job to complete successfully, ensuring no errors occur.
 View the processed output using hdfs dfs -cat output_wa/part-r-00000 to check the analysed weather

 Understanding how **MapReduce** processes large datasets by dividing tasks into mapping and reducing phases for efficient computation.

 Applying **MapReduce** to weather analysis by classifying days as shiny or cool based on temperature data.

EXPERIMENT 8

MapReduce is an efficient framework for processing large-scale datasets in parallel. It follows a **divide and conquer** approach, distributing tasks across multiple nodes for faster execution. In this experiment, MapReduce is used to find the tags associated with each movie by analyzing the **MovieLens dataset**.

Processing Movie Data with MapReduce
The dataset typically contains movie records in the format:
userId, MovieId, tag, timestamp
1, 100, Action, 1256325400
Each movie's tags are processed through **Mapper, Shuffle & Sort, and Reducer phases**.

Mapper Function
The **mapper reads movie records** from the dataset and extracts key-value pairs in the format **(userId, movieId, tag, timestamp)**. These key-value pairs are then sent to the next phase for aggregation.

Shuffle and Sort Phase
Hadoop groups movie tags **based on MovieID**, ensuring that all tags associated with the same movie are processed together. Sorting may also occur to optimize data organization.

Reducer Function
The **reducer aggregates tags** for each movie, forming a comma-separated list of tags. The output format is **(movieId, tag)**, providing a comprehensive view of movie categorization.

Execution in Hadoop
The movie dataset is stored in **HDFS**, allowing distributed processing across multiple nodes. The Hadoop job is executed using the **hadoop jar** command, and the final output—mapping movies to their associated tags—is stored in an **HDFS directory**.

By leveraging **MapReduce**, the process becomes scalable and efficient, enabling insights into **movie trends, genres, and user-generated metadata**.

- Create MovieTagsAnalysis.java using the nano text editor in the terminal.
- Enter the MapReduce program code to analyse movie tags, then save and exit.
- Compile MovieTagsAnalysis.java using javac to generate the required class files.
- Package the compiled class files into a JAR file using the jar command.
- Create a tags.csv file using nano to provide input data for the Hadoop job.
- Enter userId,-movieId-tag-timestamp data into the file, then save and exit.
- Run the Hadoop job using hadoop jar mta.jar MovieTagsAnalysis tags.csv output_mta to process the text file.
- Wait for the Hadoop job to complete successfully, ensuring no errors occur.
- View the processed output using hdfs dfs -cat output_mta/part-r-00000 to check the analysed movie tags

 Understand how **MapReduce** processes large datasets by dividing tasks across multiple nodes for

efficient computation and how key-value pairs are handled in **Mapper, Shuffle-Sort, and Reducer** phases.

 Gain practical knowledge of **Hadoop's Mapper and Reducer** for movie tag analysis. Learnt how tags are grouped for each movie, improving categorization and retrieval in large datasets.

Experiment – 9

MapReduce is a framework that enables distributed processing of large datasets by splitting tasks into smaller units executed across multiple nodes. It operates in three main phases: **Mapper**, which processes input data and emits key-value pairs; **Shuffle and Sort**, which groups similar keys together; and **Reducer**, which aggregates and computes the final results. This model is widely used in big data analysis to extract meaningful insights efficiently.

Uber Trip Analysis with MapReduce
The Uber dataset contains details about trips made by different bases on specific dates. The goal is to identify which days had the highest number of trips for each base using MapReduce.

Mapper Function

- Reads input data and extracts **dispatching base number** and **trip details**.
- Emits (BaseID, Date, Trips) as key-value pairs for further processing.

Shuffle and Sort Phase

- Groups all records belonging to the same **dispatching base number** together.
- Sorts trip counts for each date to facilitate identification of the highest trip day.

Reducer Function

- Processes grouped data to find the **date with maximum trips** for each base.
- Outputs (BaseID, Date with Highest Trips) as the final result.

Execution and Scalability

- The dataset is stored in **HDFS**, ensuring distributed and parallel execution.
- The final output is saved in an **HDFS directory**, making it accessible for further analysis.

By leveraging MapReduce, Uber trip analysis becomes efficient, scalable, and suitable for handling large transportation datasets.

 Create UberTripsAnalysis.java using the nano text editor in the terminal.
 Enter the MapReduce program code to analyse the Uber trip data, then save and exit.
 Compile UberTripsAnalysis.java using javac to generate the required class files.
 Package the compiled class files into a JAR file using the jar command.
 Create a TripsData.txt file using nano to provide input data for the Hadoop job.
 Enter base.-date-vehicle-trips data into the file, then save and exit.
 Run the Hadoop job using hadoop jar uta.jar UberTripsAnalysis TripsData.txt output_uta to process the text file.
 Wait for the Hadoop job to complete successfully, ensuring no errors occur.

 View the processed output using hdfs dfs -cat output_uta/part-r-00000 to check the analysed result

 Understand the **MapReduce framework** for distributed data processing and its application in analyzing large-scale transportation datasets like Uber trips.

 Gain insights into **data aggregation and filtering** by identifying the day with the highest trips for each dispatching base using key-value pair processing.

Experiment -10

MapReduce is a powerful framework for processing large datasets in parallel. It distributes tasks across multiple nodes, ensuring faster and efficient execution. In this experiment, MapReduce is used to analyze the Titanic dataset, determining the **average age of deceased passengers** and the **number of survivors in each class**.

Processing Titanic Data with MapReduce
The dataset consists of passenger records in the format:
Survived, Pclass, Sex, Age
0, 3, male, 22
MapReduce processes this data to extract relevant insights on survival and age distribution.

Mapper Function
The mapper reads passenger records, extracting key-value pairs. For deceased individuals, it outputs (Deceased_Gender, Age), and for survivors, it outputs (Survived_Class, 1). These key-value pairs are passed to the next phase.

Shuffle and Sort Phase
Hadoop groups records based on keys. All deceased passengers' ages are combined to calculate the **average age**, while survivor counts are grouped by class to determine how many survived in each. Sorting optimizes processing efficiency.

Reducer Function
The reducer computes the **average age** of deceased passengers by summing their ages and dividing by the count. It also aggregates the **total survivors per class**, ensuring structured and insightful output.

Execution in Hadoop
The Titanic dataset is stored in **HDFS**, allowing distributed processing across multiple nodes. The Hadoop job is executed using the hadoop jar command, and the final output—showing **deceased passengers' average age and survivor count by class**—is stored in an HDFS directory.

 Create TitanicAgeAnalysis.java using the nano text editor in the terminal.
 Enter the MapReduce program code to calculate the average age, then save and exit.
 Compile TitanicAgeAnalysis.java using javac to generate the required class files.
 Package the compiled class files into a JAR file using the jar command.
 Create a age.csv file using nano to provide input data for the Hadoop job.
 Enter Survived-Pclass-Sex-Age data into the file, then save and exit.
 Run the Hadoop job using hadoop jar taa.jar TitanicAgeAnalysis age.csv taa_output to process the text file.

 Wait for the Hadoop job to complete successfully, ensuring no errors occur.
 View the processed output using hdfs dfs -cat taa_output/part-r-00000 to check the analysed result

 Understand the **MapReduce framework** and its ability to process large datasets in a distributed environment, ensuring efficient data analysis.

 Analyze the **Titanic dataset** using MapReduce to compute the **average age of deceased passengers** and **survivor count per class**, gaining insights into structured data processing.

Experiment -11

MapReduce is a powerful framework for processing large datasets in parallel. It uses a divide-and-conquer approach by distributing tasks across multiple nodes. In this experiment, MapReduce is used to find the maximum temperature recorded each month for every year from a weather dataset.

Processing Weather Data with MapReduce
The dataset is structured in the format:
Year, Month, Temperature
2010 01 28
Each row denotes a recorded temperature for a specific month and year.

Mapper Function
The mapper reads each record, extracting the year as the key, and combines the month and temperature as the value. The key-value format is (Year, Month Temperature). These pairs are passed to the next phase.

Shuffle and Sort Phase
Hadoop groups all records by year so that the reducer receives all temperatures associated with each year. Sorting helps organize data better, aiding efficient reduction.

Reducer Function
The reducer iterates through all months and temperatures for a year, comparing each to find the highest temperature. It then emits the year and the month in which this maximum temperature occurred.

Execution in Hadoop
The dataset is uploaded to HDFS for distributed processing. The job is run using the hadoop jar command. The final output contains year-wise records of the month with the highest temperature, stored in an HDFS directory.

 Create MaxTemperatureMonth.java using the nano text editor in the terminal.
 Enter the MapReduce program code to analyse the max. temperature, then save and exit.
 Compile MaxTemperatureMonth.java using javac to generate the required class files.
 Package the compiled class files into a JAR file using the jar command.
 Create a maxtemp.txt file using nano to provide input data for the Hadoop job.
 Enter Year-Month-Temperature data into the file, then save and exit.

 Run the Hadoop job using hadoop jar mtm.jar MaxTemperatureMonth maxtemp.txt output_mtm to process the text file.
 Wait for the Hadoop job to complete successfully, ensuring no errors occur.
 View the processed output using hdfs dfs -cat output_mtm/part-r-00000 to check the maximum temperature.

- I understood how to use the MapReduce framework to process large-scale climate datasets by extracting and comparing temperature data across different months and years, which enhanced my skills in distributed data processing.

- I learned how to design and implement custom Mapper and Reducer logic to analyze patterns in weather data, specifically to determine the maximum temperature for each month within each year, improving my understanding of Hadoop-based analytics.

Experiment -12

MapReduce is a parallel data processing framework ideal for large-scale datasets. It distributes data and tasks across cluster nodes, enabling faster and scalable computations. In this experiment, MapReduce is used to **sort and aggregate word counts** from a text dataset, helping analyze word frequency efficiently.

Processing Text Data with MapReduce

The dataset contains lines of text like:
Data science is powerful and scalable
Each word is extracted, counted, and aggregated using MapReduce phases.

Mapper Function

The mapper reads each line and breaks it into words using a tokenizer. For every word, it emits (word, 1), representing a single occurrence. These intermediate pairs are sent to the next phase.

Shuffle and Sort Phase

This phase automatically sorts the keys (words) in lexicographical order before they reach the reducer. For example, if the words are science, data, powerful, they are sorted as data, powerful, science. Sorting ensures ordered processing and improves efficiency when aggregating values.

Reducer Function

The reducer receives each unique sorted word and a list of values (1s). It sums the values to get the total count of that word. The final output is (word, total_count), and the output remains **sorted by default** due to Hadoop's shuffle and sort mechanism.

Execution in Hadoop

The input file is first uploaded to **HDFS**. The Hadoop job is executed using the hadoop jar command. The output is stored in an HDFS directory as a **sorted list of words with their aggregated counts**.

 Create SortAggregate.java using the nano text editor in the terminal.

 Enter the MapReduce program code to sort and aggregate the data, then save and exit.
 Compile SortAggregate.java using javac to generate the required class files.
 Package the compiled class files into a JAR file using the jar command.
 Create a SortData.txt file using nano to provide input data for the Hadoop job.
 Enter Year-Month-Temperature data into the file, then save and exit.
 Run the Hadoop job using hadoop jar sa.jar SortAggregate SortData.txt output_sa to process the text file.
 Wait for the Hadoop job to complete successfully, ensuring no errors occur.
 View the processed output using hdfs dfs -cat output_sa/part-r-00000 to check the maximum temperature.

- Understood how MapReduce efficiently performs word-level aggregation while ensuring automatic sorting of keys during the shuffle and sort phase for structured output.

- Gained practical experience in developing and executing a MapReduce program to count word frequencies and analyze sorted textual data in a distributed environment.