PG Department of Data Science

Bishop Heber College (Autonomous),

Tiruchirappalli-620017, TamilNadu

Ranked **34th** at the National Level by MHRD through **NIRF 2023**

(Nationally Re-accredited at the **A++** Grade by NAAC with CGPA of **3.69 out of 4**)

BONAFIDE CERTIFICATE

Name : _____

Register No : _____

Class : **II M.Sc Data Science**

Course Code : **P23DS3P6**

Course Title : **Big Data Management and Analytics Lab**

Semester : **III**

Certified that this is the bonafide record of work done by me during Odd Semester of 2024-2025 and submitted to the Practical Examination on _____

Staff In-Charge **Head of the Department**

Examiners

1._____

2._____

INDEX

Exercise 1: Basic File and Directory Operations in Hadoop HDFS

Aim:

To perform basic file and directory operations in Hadoop HDFS, including creating directories, uploading files, listing directory contents, reading files, and deleting files and directories.

Procedure:

- **Start Hadoop Services**:
  Before performing any operations in HDFS, ensure that all the necessary Hadoop services are started, including the NameNode, DataNode, and ResourceManager.

- **Create a Directory in HDFS**:
  Use the appropriate command to create a directory within the Hadoop Distributed File System (HDFS). This directory will be used to store files for further operations.

- **Upload Files to HDFS**:
  Upload files from the local file system to the newly created directory in HDFS. This allows the files to be distributed across the HDFS storage nodes.

- **List Files and Directories in HDFS**:
  List the files and subdirectories within the specified HDFS directory to verify the content.

- **View the Content of a File in HDFS**:
  Display the content of a file stored in HDFS to ensure that the data is correctly uploaded.

- **Copy Files from HDFS to Local File System**:
  Retrieve a file from HDFS back to the local file system. This is useful for downloading files after processing in Hadoop.

- **Delete Files from HDFS**:
  Delete a specific file from the HDFS directory once it is no longer needed.

- **Delete Directories from HDFS**:
  Recursively delete a directory and all its contents from HDFS.

- **Stop Hadoop Services** :
  After performing the necessary file and directory operations, stop the Hadoop services to free up system resources. This step is optional depending on the environment.

Program:

 # Start Hadoop services
start-all.sh

 # Create a new directory in HDFS
hdfs dfs -mkdir /user/cloudera/my_directory

 # Upload a file from local filesystem to HDFS
hdfs dfs -put /home/cloudera/sample.txt /user/cloudera/my_directory/

 # List files and directories in HDFS
hdfs dfs -ls /user/cloudera/my_directory

 # View the content of a file in HDFS
hdfs dfs -cat /user/cloudera/my_directory/sample.txt

 # Copy the file from HDFS to local filesystem
hdfs dfs -get /user/cloudera/my_directory/sample.txt /home/cloudera/

 # Delete a file from HDFS
hdfs dfs -rm /user/cloudera/my_directory/sample.txt

 # Delete the directory in HDFS recursively
hdfs dfs -rm -r /user/cloudera/my_directory

 # Stop Hadoop services (optional)
stop-all.sh

Result:

Basic HDFS operations including creating directories, uploading files, listing, viewing, copying, and deleting files and directories were performed successfully.

Exercise: 1.1 Basic File and Directory Operations in Hadoop HDFS

- Create a directory called **student_data** in your HDFS home directory.

- Upload a file **records.txt** from local computer to the student_data directory in HDFS.

- List all files and directories inside the ***student_data*** directory to confirm the file upload.

- Display the contents of the ***records.txt*** file in HDFS.

- Copy the file ***records.txt*** from the HDFS student_data directory to your local system's Downloads folder.

- Rename the ***records.txt*** file in HDFS to ***student_records.txt.***

- Create a new directory in HDFS called ***backup*** and move **student_records.txt** from student_data to backup.

- Delete the ***student_data*** directory from HDFS.

- Remove the file ***student_records.txt*** from the backup directory in HDFS.

- Delete the backup directory from HDFS.

Exercise 2: Implementing Word Count using Hadoop MapReduce

Aim:

To learn how to implement a Word Count program using Hadoop MapReduce in Python, demonstrating data processing using the mapper and reducer design patterns, while using Cloudera's environment.

Procedure:

- **Start Cloudera Services:**
  - Open Cloudera Manager and ensure that services like HDFS and YARN are running.
- **Create Three Files and Type the corresponding content**
  - **input.txt**
  - **mapper.py**
  - **reducer.py**
- Upload the input file to HDFS
  - hdfs dfs -mkdir -p /user/cloudera/wordcount/input
  - hdfs dfs -put input.txt /user/cloudera/wordcount/input/
- **Make the Mapper and Reducer Scripts Executable:**
  - chmod +x mapper.py
  - chmod +x reducer.py
- To run the mapper.py script directly on an input file like input.txt, you can use the command line.
  - cat input.txt | python mapper.py **➔Refer Output-1**
- To aggregate these counts, you'll need to pass this output to the reducer.
  - cat input.txt | python mapper.py | sort | python reducer.py **➔Refer Output-2**

Program:

input.txt

Hadoop is great
Hadoop is scalable
Hadoop is open-source

mapper.py

```
#!/usr/bin/env python3
import sys
def mapper():
    for line in sys.stdin:
        line = line.strip()  # Remove leading/trailing whitespace
        words = line.split()  # Split the line into words
        for word in words:
            print("%s\t%d" % (word, 1))  # Output word with a count of 1

if __name__ == "__main__":
    mapper()
```

reducer.py

```
#!/usr/bin/env python3
import sys
def reducer():
    current_word = None
    current_count = 0

    for line in sys.stdin:
        line = line.strip()
        word, count = line.split('\t')
        try:
            count = int(count)
        except ValueError:
            continue

        if current_word == word:
            current_count += count
        else:
            if current_word:
              print("%s\t%d" % (current_word, current_count))
            current_word = word
            current_count = count

    if current_word == word:
        print("%s\t%d" % (current_word, current_count))

if __name__ == "__main__":
    reducer()
```

Output-1: cat input.txt | python mapper.py

```
Hadoop      1
is          1
great       1
Hadoop      1
is          1
scalable    1
Hadoop      1
is          1
open-source 1
```

Output-2: cat input.txt | python mapper.py | sort | python reducer.py

```
Hadoop      3
is          3
great       1
scalable    1
open-source 1
```

Result:

The Word Count program was successfully implemented using Hadoop MapReduce in Python on Cloudera. The program reads the input file, counts the occurrences of each word, and outputs the results using the format specifier method for string formatting.

Exercise 3: Implementing Word Count by skip the stop words using Hadoop MapReduce

Aim:

To implement a Word Count program using Hadoop MapReduce in Python that skips common stop words. This program will count the occurrences of each word from the input data while excluding words from a predefined list of stop words, demonstrating how to filter irrelevant terms from a dataset using the MapReduce paradigm.

Procedure:

- **Start Cloudera Services:**
  - Open Cloudera Manager and ensure that services like HDFS and YARN are running.
- **Create Three Files and Type the corresponding content**
  - **input.txt**
  - **mapper.py**
  - **reducer.py**
- Upload the input file to HDFS
  - hdfs dfs -mkdir -p /user/cloudera/wordcount/input
  - hdfs dfs -put input.txt /user/cloudera/wordcount/input/
- **Make the Mapper and Reducer Scripts Executable:**
  - chmod +x mapper.py
  - chmod +x reducer.py
- To run the mapper.py script directly on an input file like input.txt, you can use the command line.
  - cat input.txt | python mapper.py **➔Refer Output-1**
- To aggregate these counts, you'll need to pass this output to the reducer.
  - cat input.txt | python mapper.py | sort | python reducer.py **➔Refer Output-2**

Program:

input.txt

```
Hadoop is great
Hadoop is scalable
Hadoop is open-source
```

mapper.py

```
#!/usr/bin/env python3
import sys
# Define stop words as a list directly in the script
stopwords = ["is", "a", "the", "for", "and", "of", "to", "in", "on", "with", "by", "it"]

def mapper():
    for line in sys.stdin:
        line = line.strip().lower()  # Convert to lowercase for consistency
        words = line.split()  # Split the line into words

        for word in words:
            if word not in stopwords:  # Skip the stop words
                print("%s\t%d" % (word, 1))  # Output word with a count of 1

if __name__ == "__main__":
    mapper()
```

reducer.py

```
#!/usr/bin/env python3
import sys
def reducer():
    current_word = None
    current_count = 0

    for line in sys.stdin:
        line = line.strip()
        word, count = line.split('\t')
        try:
            count = int(count)
        except ValueError:
            continue

        if current_word == word:
            current_count += count
        else:
            if current_word:
                print("%s\t%d" % (current_word, current_count))
            current_word = word
            current_count = count

    if current_word == word:
        print("%s\t%d" % (current_word, current_count))

if __name__ == "__main__":
    reducer()
```

Output-1: cat input.txt | python mapper.py

```
Hadoop      1
great       1
Hadoop      1
scalable    1
Hadoop      1
open-source 1
```

Output-2: cat input.txt | python mapper.py | sort | python reducer.py

```
Hadoop      3
great       1
scalable    1
open-source 1
```

Result:

The mapper will process the input, ignore the words in the stop words list, and count the occurrences of the remaining words. The final output will exclude common stop words.

Exercise 3.1: Count the occurrences of the List of Keywords from given text files using Hadoop Mapreduce

input-data.txt

Hadoop is an open-source framework for data storage and large-scale data processing. It provides high availability and fault tolerance in distributed environments. Organizations use Hadoop for handling massive amounts of structured and unstructured data.

Keywords

- Hadoop, MapReduce, data, framework

mapper.py

reducer.py

Exercise 4: Loading and Complex Data Transformations in Pig

Aim:

To load large datasets into Pig, perform complex data transformations, including filtering, joining, grouping, and advanced aggregations, and apply multiple filtering conditions.

Procedure:

1. Start the Pig CLI or Grunt shell.
2. Load the datasets **employee_data.csv** and **department_data.csv** into Pig.
3. Filter the records based on salary and department conditions.
4. Perform a join between two datasets on a common key.
5. Group the joined data by department.
6. Calculate advanced aggregations such as total and average salary, and employee count per department.
7. Apply an additional filter on the grouped data.
8. Store the final result in a new dataset.

Program:

```
-- Step 1: Load the employee dataset
employee_data = LOAD 'employee_data.csv' USING PigStorage(',')
   AS (emp_id:int, emp_name:chararray, department_id:int, salary:float, age:int);

-- Step 2: Load the department dataset
department_data = LOAD 'department_data.csv' USING PigStorage(',')
   AS (department_id:int, department_name:chararray);

-- Step 3: Filter employees with salary greater than 70,000 and age greater than 30
filtered_employees = FILTER employee_data BY salary > 70000 AND age > 30;

-- Step 4: Join the employee data with department data based on department_id
joined_data = JOIN filtered_employees BY department_id, department_data BY department_id;

-- Step 5: Group the joined data by department
grouped_by_department = GROUP joined_data BY department_data::department_name;

-- Step 6: Calculate total salary, average salary, and number of employees per department
department_aggregates = FOREACH grouped_by_department GENERATE
   group AS department_name,
   COUNT(joined_data) AS employee_count,
   SUM(joined_data.salary) AS total_salary,     AVG(joined_data.salary) AS avg_salary;
```

-- Step 7: Filter out departments with fewer than 2 employees
filtered_departments = FILTER department_aggregates BY employee_count >= 2;

-- Step 8: Store the result in a new file filtered_department_summary
STORE filtered_departments INTO 'filtered_department_summary' USING PigStorage(',');

Sample Datasets:

employee_data.csv (50 records)

emp_id	emp_name	department_id	Salary	age
1	John Doe	101	85000	35
2	Jane Smith	102	72000	45
3	David Brown	101	95000	40
4	Mary Johnson	103	60000	29
5	Michael Lee	102	68000	32
6	Alice White	101	73000	28
7	Robert Green	104	55000	36
8	Susan Black	103	78000	33
9	James Davis	104	51000	25
10	Linda Clark	101	99000	38
...				

department_data.csv (10 records)

department_id	department_name
101	Engineering
102	Marketing
103	Sales
104	HR
105	Operations

Output:

- **Filtered employees with salary greater than 70,000 and age greater than 30**:

emp_id	emp_name	department_id	Salaryage	
1	JohnDoe	101	85000	35
2	JaneSmith	102	72000	a45
3	DavidBrown	101	95000	40
10	LindaClark	101	99000	38

- **Aggregated department data (total salary, average salary, and employee count)**:

department_name	employee_count	total_salary	avg_salary
Engineering	3	279000	93000.0
Marketing	1	72000	72000.0

- **Filtered department data with at least 2 employees**:

department_name	employee_count	total_salary	avg_salary
Engineering	3	279000	93000.0

Result:

Successfully performed complex data transformations in Pig, including multi-condition filtering, joining datasets, grouping by department, and calculating aggregate metrics. The final filtered data was stored in a new file for analysis.

Exercise-4.1: Loading and Complex Data Transformations in Pig

Dataset

- **customers.csv**{ customer_id,name,location}
- **transactions.csv**:{ transaction_id,customer_id,item,amount}

Tasks

- **Load the datasets**:
  - Load the customers.csv and transactions.csv files into Pig.

- **Filter Transactions**:
  - Filter out transactions where greater than or equal to $1000.

- **Group transactions by customer**:
  - Group the filtered transactions by customer_id.

- **Join customer and transaction data**:
  - Perform a join between the customers and the filtered transactions on customer_id to include customer details (name and location) in the results.

- **Calculate total spending per customer**:
  - For each customer, calculate the total amount they have spent.

- **Find top 3 spenders**:
  - Sort the customers based on the total amount spent and retrieve the top 3 customers.

- **Group by location and calculate total spending per location**:
  - Group the customers by their location and calculate the total amount spent by customers in each location on high-value items.

Output

Exercise 5: Advanced Data Transformations using Pig

Aim:

To learn how to perform advanced data transformations using Pig, including joins and grouping operations.

Procedure:

- Start the Pig CLI.
- Load the dataset into Pig with the appropriate schema.
- Filter the dataset to include only relevant records.
- Group the data by a specified attribute.
- Perform joins between two datasets based on a common key.
- Execute aggregate functions to analyze the grouped data.
- Store the results in a new dataset.

Program:

```
-- Load the product data
product_data = LOAD 'product_data.csv' USING PigStorage(',')
   AS (product_id:chararray, product_name:chararray, price:float, category:chararray);

-- Load the sales data
sales_data = LOAD 'sales_data.csv' USING PigStorage(',')
   AS (order_id:int, product_id:chararray, quantity:int, order_date:chararray);

-- Filter products in the 'Electronics' category
filtered_products = FILTER product_data BY category == 'Electronics';

-- Group sales data by product_id
grouped_sales = GROUP sales_data BY product_id;

-- Join filtered products with grouped sales
joined_data = JOIN filtered_products BY product_id, grouped_sales BY product_id;

-- Calculate total sales per product
total_sales = FOREACH joined_data GENERATE
   filtered_products::product_name,
   SUM(sales_data.quantity * filtered_products.price) AS total_sales;

-- Store the result in a new dataset
STORE total_sales INTO 'total_sales_data' USING PigStorage(',');
```

Sample Dataset:

product_data.csv

product_id	product_name	price	Category
P001	Laptop	500.0	Electronics
P002	Smartphone	300.0	Electronics
P003	Refrigerator	700.0	Appliances
P004	Headphones	100.0	Electronics
P005	Microwave	150.0	Appliances

sales_data.csv

order_id	product_id	quantity	order_date
1	P001	2	2024-01-15
2	P002	1	2024-02-05
3	P003	1	2024-01-25
4	P004	5	2024-03-12
5	P005	3	2024-01-30

Output:

- **Filtered Products in Electronics Category:**

product_id	product_name	price	Category
P001	Laptop	500.0	Electronics
P002	Smartphone	300.0	Electronics
P004	Headphones	100.0	Electronics

- **Total Sales per Product:**

product_name	total_sales
Laptop	1000.0
Smartphone	300.0
Headphones	500.0

Result:

Successfully executed advanced data transformations in Pig, including filtering, grouping, joining datasets, and calculating total sales for products in the Electronics category. The results were stored in a new dataset for further analysis.

Exercise 5.1: Advanced Data Transformations using Pig

Dataset

- **products.csv**:{ product_id,product_name,category,price}
- **sales.csv:{** sale_id,product_id,customer_id,quantity,sale_date}

Tasks

- **Load the Datasets**:
  - Load the products.csv and sales.csv files into Pig using the appropriate loader.

- **Calculate Total Revenue Per Sale**:
  - For each sale, calculate the total revenue by multiplying the quantity sold by the product price.

- **Filter Sales for High-Value Products**:
  - Filter out sales where the product price is greater than or equal to $500.

- **Group Sales by Category**:
  - Group the sales by product category and calculate the total quantity sold and total revenue for each category.

- **Join Products and Sales Data**:
  - Perform a join between the products and sales datasets based on the product ID to enrich the sales data with product details.

- **Calculate Top 3 Products by Revenue**:
  - For each product, calculate the total revenue and then sort the products based on total revenue. Extract the top 3 products.

- **Identify Product Categories with Sales Above a Threshold**:
  - Filter out categories that have total sales revenue greater than $1000.

Output:

Exercise 6: Creating and Querying Tables in Hive with SQL

Aim:

To learn how to create tables in Hive, load data into tables, and perform basic SQL queries such as SELECT, WHERE, and GROUP BY.

Procedure:

- Start the Hive CLI.
- Create a new database to organize your tables and switch to it.
- Create a new table to store sales data with appropriate columns and data types.
- Load the data into the sales_data table from an HDFS location.
- Run SQL queries to analyze the data, including displaying all rows, calculating total sales per product, and filtering data by category.

Program:

```
CREATE DATABASE IF NOT EXISTS sales_db;
USE sales_db;

CREATE TABLE sales_data (
order_id INT,
product_name STRING,
category STRING,
price FLOAT,
quantity INT,
order_date STRING
) ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE;

LOAD DATA INPATH 'sales_data.csv' INTO TABLE sales_data;

-- Display all data
SELECT * FROM sales_data;

-- Calculate total sales per product
SELECT product_name, SUM(price * quantity) AS total_salesFROM sales_dataGROUP BY
product_name;

-- Filter data for 'Electronics' category
SELECT * FROM sales_dataWHERE category = 'Electronics';
```

Sample Dataset:

- **sales_data.csv**{order_id,product_name,category,price,quantity,order_date}
  1,Laptop,Electronics,500.0,2,2024-01-15
  2,Smartphone,Electronics,300.0,1,2024-02-05
  3,Refrigerator,Appliances,700.0,1,2024-01-25
  4,Headphones,Electronics,100.0,5,2024-03-12
  5,Microwave,Appliances,150.0,3,2024-01-30

Output:

- **Displaying all rows from the table:**

order_id	product_name	category	price	quantity	order_date
1	Laptop	Electronics	500.0	2	2024-01-15
2	Smartphone	Electronics	300.0	1	2024-02-05
3	Refrigerator	Appliances	700.0	1	2024-01-25
4	Headphones	Electronics	100.0	5	2024-03-12
5	Microwave	Appliances	150.0	3	2024-01-30

- **Total sales per product:**

product_name	total_sales
Laptop	1000.0
Smartphone	300.0
Refrigerator	700.0
Headphones	500.0
Microwave	450.0

- **Filtered data for the 'Electronics' category:**

order_id	product_name	category	price	quantity	order_date
1	Laptop	Electronics	500.0	2	2024-01-15
2	Smartphone	Electronics	300.0	1	2024-02-05
4	Headphones	Electronics	100.0	5	2024-03-12

Result:

Successfully created a table in Hive, loaded sales data, and performed various SQL queries to analyze the data.

Exercise 6.1: Creating and Querying Tables in Hive with SQL

DataSet

- **products.csv**:{product_id, product_name, category, price} : 5Records
- **sales.csv**:{sale_id, product_id, customer_id, quantity, sale_date} :10Records

Tasks

- **Start the Hive CLI.**
- **Create a Database.**
- **Use the Created Database**
- **Create a Table for Products{product_id, product_name, category, price}**
- **Load Data into Products Table.**
- **Create a Table for Sales.**
- **Load Data into Sales Table.**

- **Query to Select All Products.**

- **Query to Get Products with Price Greater than $500.**

- **Query to Count Total Sales by Product.**

- **Query to Get Total Revenue by Product.**

Output

Output

Exercise 7: Implementing Partitioning in Hive and Querying Partitioned Data

Aim:

To understand and apply partitioning in Hive to improve query performance and efficiently manage large datasets by partitioning based on a specific column, such as course.

Procedure:

- Start the Hive CLI.
- Create a new database to organize your tables and switch to it.
- Create a partitioned table to store student data, where the data is partitioned by the course column.
- Load data into the partitioned table, specifying the partition values for each course.
- Run queries on the partitioned table to verify that partitioning improves query performance and to check if partitioned data can be accessed correctly.

Program:

```
-- Create a database
CREATE DATABASE IF NOTEXISTSstudent_db;
USE student_db;

-- Create a partitioned table
CREATETABLEstudent_data (
student_idINT,
student_name STRING,
ageINT,
grade STRING
)
PARTITIONED BY (course STRING)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY','
STORED AS TEXTFILE;

-- Load data into the partitioned table
-- You need to specify the partition value when loading data
ALTER TABLE  student_data  ADD  PARTITION (course='Mathematics');
```

```
LOAD DATA INPATH 'student_data_mathematics.csv' INTO TABLE student_data
PARTITION (course='Mathematics');

ALTER TABLE  student_data  ADD  PARTITION (course='Science');
LOAD DATA INPATH '/user/hive/student_data_science.csv' INTO TABLE student_data
PARTITION (course='Science');
```

--Display available partitions

```
Show partitions student_data
```

-- Run queries on the partitioned table

-- Display all data from a specific partition (e.g., Mathematics)

```
SELECT*FROMstudent_dataWHERE course ='Mathematics';
```

-- Query across all partitions to get the count of students per course

```
SELECT course, COUNT(student_id) ASstudent_countFROMstudent_dataGROUPBY
course;
```

-- Filter data for students above a certain age across all partitions

```
SELECT*FROMstudent_dataWHERE age >20;
```

Sample Dataset:

- **student_data_mathematics.csv**:{student_id,student_name,age,grade}

  101,John Doe,22,A
  102,Jane Smith,21,B

- **student_data_science.csv**:{student_id,student_name,age,grade}

  201,Emily Davis,23,A
  202,Michael Brown,24,B

Output:

- **Available Partitions**

  Course=Mathematics
  Course=Science

- **Displaying all data from a specific partition (e.g., Mathematics):**

student_id	student_name	age	grade
101	John Doe	22	A
102	Jane Smith	21	B

- **Count of students per course across all partitions:**

course	student_count
Mathematics	2
Science	2

- **Filtered data for students above age 20 across all partitions:**

student_id	student_name	age	grade	course
101	John Doe	22	A	Mathematics
102	Jane Smith	21	B	Mathematics
201	Emily Davis	23	A	Science
202	Michael Brown	24	B	Science

Result:

Successfully created a partitioned table in Hive, loaded student data into specific partitions based on the course column, and performed queries to verify that partitioning improves query performance and data management.

Exercise 7.1: Implementing Partitioning in Hive and Querying Partitioned Data

DataSet

- **products-electronics.csv**
  1,Laptop,1000
  2,Smartphone,700
  3,Tablet,400
  4,Monitor,300
  5,TV,1200
- **products-wearables.csv**:
  6,Smartwatch,200
  7,Fitness Tracker,250
  8,Wireless Earbuds,150
  9,VR Headset,400
  10,Smart Ring,300

Tasks

- **Start the Hive CLI**.
- **Create a Database**.
- **Use the Created Database**.
- **Create a Partitioned Table for Products { product_id INT, product_name STRING, price FLOAT} with partion category**
- **Add two partitions Electronics and Wearable**
- **Load Electronics Data into the Partitioned Table.**

- **Load Wearable Data into the Partitioned Table.**

- **Query to Select All Products.**

- **Query to Filter Products by Category.**

- **Query to Get Average Price by Category.**

Output

Exercise 8: Implement Indexing on a Hive table and query indexed data

Aim:

To demonstrate the process of creating an indexed table in Hive and querying data from it to understand the benefits of indexing for query optimization. This exercise includes creating an index on the Hive table, running queries on indexed data, and comparing the performance before and after indexing.

Procedure:

- Start the Hive CLI
- Create a new database
- Create a table and load data into it
- Create an index on the table
- Rebuild the index
- Run queries on the indexed table
- Display query execution plan
- Drop the index and table (cleanup)

Program

```
-- Step 1: Create a database
CREATE DATABASE IF NOT EXISTS demo_db;
USE demo_db;

-- Step 2: Create a table
CREATE TABLE student_data (
        student_id INT,  student_name STRING,  course STRING,  grade STRING)
        ROW FORMAT DELIMITED
        FIELDS TERMINATED BY ','
        STORED AS TEXTFILE;

-- Step 3: Load data into the table
LOAD DATA LOCAL INPATH 'student_data.csv' INTO TABLE student_data;

-- Step 4: Create an index on the 'course' column
CREATE INDEX idx_course  ON TABLE student_data (course) AS 'COMPACT';
```

-- Step 5: Rebuild the index
ALTER INDEX idx_course ON student_data REBUILD;

-- Step 6: Query to select data from the indexed table
SELECT * FROM student_data WHERE course = 'Mathematics';

-- Step 7: Display query execution plan using EXPLAIN
EXPLAIN SELECT * FROM student_data WHERE course = 'Mathematics';

-- Step 8: Drop the index and the table (cleanup)
DROP INDEX IF EXISTS idx_course ON student_data;
DROP TABLE IF EXISTS student_data;

Data File: student_data.csv:

student_id	student_name	course	grade
1	John Doe	Mathematics	A
2	Jane Smith	Science	B
3	Emily Davis	Mathematics	C
4	Michael Brown	History	A
5	Lucas White	Mathematics	B
6	Anna Johnson	Science	A
7	Paul Walker	History	B
8	Emma Wilson	Mathematics	A
9	Olivia Brown	Science	C
10	James Smith	History	A

Output

❖ **Query Results for Indexed Table**: Students enrolled in "Mathematics":

student_id	student_name	Course	grade
1	John Doe	Mathematics	A
3	Emily Davis	Mathematics	C
5	Lucas White	Mathematics	B
8	Emma Wilson	Mathematics	A

❖ **Index Usage in Query Plan**: The EXPLAIN command output will show how Hive utilizes the index to optimize the query execution.

```
EXPLAIN SELECT * FROM student_data WHERE course = 'Mathematics';
 +--------------------------------------+
| Table Scan                           |
+--------------------------------------+
| Table: student_data (course)   |
| Index: idx_course                    |
| Filter: course = 'Mathematics' |
+--------------------------------------+
```

Result:

Successfully demonstrated the creation of an indexed table in Hive and observed how indexing improves query performance. The use of an index on the course column helped to optimize data retrieval when querying on this column.

Exercise-8.1: Implementing Indexing on a Hive Table and Querying Indexed Data

Dataset

- **employees.csv**:{employee_id, employee_name, department, salary}

Tasks

- **Start the Hive CLI**.
- **Create a Database** (if needed).
- **Use the Created Database**.
- **Create an Employees Table**{ employee_id INT, employee_name STRING, department STRING, salary FLOAT}
- **Load Data into the Employees Table**.
- **Create an Index on the Employees Table**.

- **Query to Select All Employees**.

- **Query to Filter Employees by Department**.

Output

Exercise 9: Performing Joins and Aggregations on Large Datasets in Hive

Aim:

To perform various types of joins and aggregations on large datasets in Hive, demonstrating how to combine and analyze data from multiple tables.

Procedure:

- Start the Hive CLI.
- Create a new database to organize your tables and switch to it.
- Create and load data into two tables: one for student information and one for course information.
- Perform different types of joins (e.g., inner join, left join) between the tables.
- Execute aggregation queries to summarize the data, such as calculating average grades or total enrollments per course.

Program:

```
-- Create a database
CREATE DATABASE IF NOTEXISTS school_db;
USE school_db;

-- Create tables and load data
CREATE TABLE students (
student_idINT,
student_name STRING,
course STRING,
grade STRING
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY','
STORED AS TEXTFILE;

CREATE TABLE courses (
course_name STRING,
instructor STRING,
creditsINT
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY','
STORED AS TEXTFILE;
```

-- Load data into the tables
LOAD DATA INPATH 'students.csv'INTOTABLE students;
LOAD DATA INPATH ‘courses.csv'INTOTABLE courses;

-- Inner Join: Get student names and course details for students enrolled in each course
SELECTs.student_name, c.course_name, c.instructorFROM students s JOIN courses c ONs.course=c.course_name;

-- Left Join: Get all students and their corresponding course details, including students not enrolled in any course
SELECTs.student_name, c.course_name, c.instructorFROMstudents sLEFTJOIN courses c ONs.course=c.course_name;

-- Aggregation: Calculate the average grade per course
SELECT course, AVG(
CASE grade
WHEN'A'THEN4
WHEN'B'THEN3
WHEN'C'THEN2
WHEN'D'THEN1
ELSE0
END
) ASaverage_grade
FROM students
GROUPBY course;

-- Aggregation: Count the number of students enrolled in each course
SELECT course, COUNT(student_id) ASstudent_countFROM students GROUPBY course;

Sample Dataset:

- **students.csv**: {student_id,student_name,course,grade}
  1,JohnDoe,Mathematics,A
  2,JaneSmith,Science,B
  3,EmilyDavis,Mathematics,C
  4,MichaelBrown,History,A
- **courses.csv**: {course_name,instructor,credits}
  Mathematics,Dr. Smith,3
  Science,Dr. Johnson,4
  History,Dr. Lee,3

Output:

- **Inner Join Result:**

student_name	course_name	instructor
John Doe	Mathematics	Dr. Smith
Jane Smith	Science	Dr. Johnson
Emily Davis	Mathematics	Dr. Smith
Michael Brown	History	Dr. Lee

- **Left Join Result:**

student_name	course_name	instructor
John Doe	Mathematics	Dr. Smith
Jane Smith	Science	Dr. Johnson
Emily Davis	Mathematics	Dr. Smith
Michael Brown	History	Dr. Lee

- **Average Grade Per Course:**

course	average_grade
Mathematics	3.0
Science	3.0
History	4.0

- **Student Count Per Course:**

course	student_count
Mathematics	2
Science	1
History	1

Result:

Successfully performed various types of joins and aggregations on large datasets in Hive. Demonstrated how to combine data from multiple tables and summarize information effectively.

Exercise 9.1: Performing Joins and Aggregations on Large Datasets in Hive

Dataset

- **employees.csv {employee_id, employee_name, department_id, salary}**
- **departments.csv{ department_id, department_name}**

Tasks

- **Start the Hive CLI**.

- **Create a Database** (if needed).

- **Use the Created Database**.

- **Create the Employees Table{ employee_id INT, employee_name STRING, department_id INT, salary FLOAT}**

- **Load Data into the Employees Table**.

- **Create the Departments Table{ department_id INT, department_name STRING}**

- **Load Data into the Departments Table**.

- **Perform a Join between Employees and Departments**.

- **Calculate Average Salary by Department**.

- **Get Total Employees per Department**.

Output

Exercise 10: Demonstrating the Use of Internal and External Tables in Hive

Aim:

To understand and demonstrate the use of internal and external tables in Hive, including how to create, manage, and query both types of tables. This exercise also covers accessing the data of an external table after it has been dropped.

Procedure:

- Start the Hive CLI.
- (Optional) Create a new database to organize your tables and switch to it.
- Create and load an internal table with data.
- Create and load an external table with data from an external location.
- Run queries to compare the behavior of internal and external tables.
- Drop the internal and external tables.
- Attempt to access the dropped tables and their data to observe the differences in behavior.

Sample Dataset:

1. **students.csv**: {student_id,student_name,course,grade}
   1,JohnDoe,Mathematics,A
   2,JaneSmith,Science,B
   3,EmilyDavis,Mathematics,C
   4,MichaelBrown,History,A

Program:

```
-- Create a database
CREATE DATABASE IF NOTEXISTSdemo_db;
USE demo_db;

-- Create an internal table
CREATETABLEinternal_table (
student_idINT,
student_name STRING,
course STRING,
grade STRING
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY','
STORED AS TEXTFILE;

-- Load data into the internal table
LOAD DATA LOCAL INPATH 'students.csv'INTOTABLEinternal_table;




-- Create an external table
CREATEEXTERNALTABLEexternal_table (
student_idINT,
student_name STRING,
course STRING,
grade STRING
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY','
LOCATION '/user/hive/external_students';

-- Load data into the external table
LOAD DATA LOCAL INPATH 'students.csv'INTOTABLEexternal_table;

-- Query internal table
SELECT*FROMinternal_table;
```

-- Query external table
SELECT*FROMexternal_table;

-- Drop the internal table (this will remove the data)
DROPTABLEinternal_table;

-- Drop the external table (this will not remove the data)
DROPTABLEexternal_table;

-- Accessing external table data directly in HDFS after dropping the table
-- Verify that the data still exists in HDFS
-- Use Hadoop commands to check the data location
dfs-ls/user/hive/external_students;

-- Read data directly from HDFS to confirm it exists
dfs-cat /user/hive/external_students/student_data.csv;

-- To reuse the data, Create an external table that refers the existing data file path
CREATE**EXTERNAL**TABLEexternal_table (
student_idINT,
student_name STRING,
course STRING,
grade STRING)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY','
LOCATION '**/user/hive/external_students**';

-- Query external table
SELECT*FROMexternal_table;

Output:

- **Query Results for Internal Table:**

student_id	student_name	course	grade
1	John Doe	Mathematics	A
2	Jane Smith	Science	B
3	Emily Davis	Mathematics	C
4	Michael Brown	History	A

- **Query Results for External Table:**

student_id	student_name	course	grade
1	John Doe	Mathematics	A
2	Jane Smith	Science	B
3	Emily Davis	Mathematics	C
4	Michael Brown	History	A

- **Read data directly from HDFS to confirm it exists**

1,JohnDoe,Mathematics,A
2,JaneSmith,Science,B
3,EmilyDavis,Mathematics,C
4,MichaelBrown,History,A

Result:

Successfully demonstrated the creation, data loading, and querying of both internal and external tables in Hive. Illustrated the key differences in data management and persistence between internal and external tables, and showed how to access data from an external table even after it has been dropped.

Exercise 10.1: Demonstrating the Use of Internal and External Tables in Hive

Dataset

- **Products.csv { product_id, product_name, category, price}**

Tasks

- **Start the Hive CLI**.

- **Create a Database** (if needed).

- **Use the Created Database**.

- **Create an pdt_internal Table{ product_id INT, product_name STRING, category STRING, price FLOAT}**

- **Load Data into the Internal Table**.

- **Query the Internal Table**.

- **Create an pdt_external Table{ product_id INT, product_name STRING, category STRING, price FLOAT}**

- **Load Data into the External Table**.

- **Query the External Table**.

- **Drop Tables (Cleanup)**.

Output