Below are the HDFS commands to perform the specified tasks. These commands are typically executed in a Hadoop environment using the Hadoop Command Line Interface (CLI):

i. Display the present working directory

```
hadoop fs -pwd
```

or

```
hdfs dfs -pwd
```

ii. Display the list of files in the Hadoop file system

```
hadoop fs -ls /
```

or

```
hdfs dfs -ls /
```

Replace / with the path to the directory you want to list.

iii. Create a new directory in the Hadoop file system

```
hadoop fs -mkdir /new_directory
```

or

```
hdfs dfs -mkdir /new_directory
```

Replace /new_directory with your desired directory path.

iv. Copy file from Hadoop to the local system

```
hadoop fs -copyToLocal /hdfs_file_path /local_directory_path
```

or

```
hdfs dfs -copyToLocal /hdfs_file_path /local_directory_path
```

Replace /hdfs_file_path with the file's path in HDFS and /local_directory_path with the destination path on your local machine.

v. Display the content of a file in the Hadoop file system

```
hadoop fs -cat /hdfs_file_path
```

or

```
hdfs dfs -cat /hdfs_file_path
```

Replace /hdfs_file_path with the file's path in HDFS.

vi. Move the file from the local system to HDFS and vice versa

Move file from local system to HDFS:

```
hadoop fs -moveFromLocal /local_file_path /hdfs_directory_path
```

or

```
hdfs dfs -moveFromLocal /local_file_path /hdfs_directory_path
```

Move file from HDFS to local system:

```
hadoop fs -moveToLocal /hdfs_file_path /local_directory_path
```

or

```
hdfs dfs -moveToLocal /hdfs_file_path /local_directory_path
```

Replace file paths and directory paths as needed.

Notes:

- Ensure you have appropriate permissions for the HDFS directories and the local system paths.
- Replace hadoop fs with hdfs dfs if you're using Hadoop 2.x or later, as hdfs dfs is the preferred syntax.

Q2

To perform the specified operations using MongoDB, follow these steps. Ensure that MongoDB is installed and running on your system, and use a MongoDB client like the Mongo Shell, MongoDB Compass, or a programming language with MongoDB drivers.

Step 1: Create the Database

Run the following command in the MongoDB shell to create a database named MCA:

```
use MCA
```

The use command creates the database if it doesn't exist and switches to it.

Step 2: Create a Collection

Create a collection named Students:

```
db.createCollection("Students")
```

Step 3: Insert Documents

a. Insert a Single Document:

```
db.Students.insertOne({
  stud_roll: 1,
  stud_fname_stud_lname: "Alice Smith",
  course: "MCA"
})
```

b. Insert Multiple Documents:

```
db.Students.insertMany([
   { stud_roll: 2, stud_fname_stud_lname: "Bob Johnson", course: "MCA" },
   { stud_roll: 3, stud_fname_stud_lname: "Charlie Brown", course: "BCA" },
   { stud_roll: 4, stud_fname_stud_lname: "David Lee", course: "BCA" }
])
```

Step 4: Find a Student by stud_roll

To find a student based on their stud_roll, use:

```
db.Students.find({ stud_roll: 1 })
```

For a formatted output:

```
db.Students.find({ stud_roll: 1 }).pretty()
```

Step 5: Update a Student's Course

a. Update a Single Document:

To update the course for a student with stud_roll 1:

```
db.Students.updateOne(
   { stud_roll: 1 },
   { $set: { course: "MBA" } }
)
```

b. Update Multiple Documents:

To update all students' courses from "BCA" to "MCA":

```
db.Students.updateMany(
   { course: "BCA" },
   { $set: { course: "MCA" } }
)
```

Step 6: Delete a Student Record

To delete records of students whose course is "BCA":

```
db.Students.deleteMany({ course: "BCA" })
```

Step 7: Verify Changes

To check the current state of the Students collection, run:

```
db.Students.find().pretty()
```

Summary of MongoDB Queries

Operation	**Command Example**
Create Database	use MCA
Create Collection	db.createCollection("Students")
Insert Single Document	db.Students.insertOne({stud_roll: 1, stud_fname_stud_lname: "Alice Smith", course: "MCA"})
Insert Multiple Documents	db.Students.insertMany([{stud_roll: 2, ...}, {stud_roll: 3, ...}])
Find by stud_roll	db.Students.find({stud_roll: 1})
Update Student Course (Single)	db.Students.updateOne({stud_roll: 1}, {$set: {course: "MBA"}})
Update Student Course (Multiple)	db.Students.updateMany({course: "BCA"}, {$set: {course: "MCA"}})
Delete Students with Course "BCA"	db.Students.deleteMany({course: "BCA"})

This set of operations allows you to manage student records effectively in MongoDB.

Q3 word count

Below is a complete Java program to perform a word count using Apache Spark. This program reads a text file, counts the occurrences of each word, and outputs the results.

Word Count Program in Java

```
import org.apache.spark.SparkConf;

import org.apache.spark.api.java.JavaRDD;

import org.apache.spark.api.java.JavaPairRDD;

import org.apache.spark.api.java.JavaSparkContext;

import scala.Tuple2;
import java.util.Arrays;


public class WordCount {
    public static void main(String[] args) {
        // Step 1: Create a Spark configuration and Spark context
        SparkConf conf = new SparkConf().setAppName("WordCount").setMaster("local[*]");
        JavaSparkContext sc = new JavaSparkContext(conf);


        // Step 2: Load the input file (path provided as argument)
        if (args.length < 1) {
            System.out.println("Usage: WordCount <input-file>");
            System.exit(1);
        }
        String inputFile = args[0];
        JavaRDD<String> input = sc.textFile(inputFile);


        // Step 3: Perform the word count
        JavaPairRDD<String, Integer> wordCounts = input
                // Split each line into words
                .flatMap(line -> Arrays.asList(line.split("\\s+")).iterator())
                // Map each word to a key-value pair (word, 1)
                .mapToPair(word -> new Tuple2<>(word, 1))
                // Reduce by key to count occurrences
                .reduceByKey(Integer::sum);


        // Step 4: Save or display the output
        String outputFile = "output";
        wordCounts.saveAsTextFile(outputFile);


        // Print the results to the console (for debugging)
        wordCounts.foreach(data -> System.out.println(data._1() + " : " + data._2()));
        // Step 5: Stop the Spark context
        sc.close();
    }
}
```

Steps to Run the Program

1. Set Up Apache Spark in Your Environment

- Install Apache Spark: Download Apache Spark.
- Set up Java Development Kit (JDK 8+).
- Add Spark and Hadoop libraries to your CLASSPATH.

2. Compile the Program

Save the program in a file named WordCount.java and compile it:

```
javac -cp "path/to/spark-core.jar:path/to/other-dependencies.jar" WordCount.java
```

3. Package the Program

Create a JAR file:

```
jar -cvf WordCount.jar WordCount*.class
```

4. Run the Program

Submit the JAR file to Spark using the spark-submit command:

```
spark-submit --class WordCount --master local[*] WordCount.jar /path/to/input.txt
```

5. Output

- The results will be saved in the output directory.
- You can check the output using:
- cat output/part-00000

Explanation of Key Components

1. **flatMap**: Splits lines into words.
2. **mapToPair**: Maps each word to a key-value pair (word, 1).
3. **reduceByKey**: Aggregates counts for each word.
4. **saveAsTextFile**: Saves the word counts to an output file.

This program demonstrates the classic word count example in Apache Spark using Java, which is efficient for large-scale text processing tasks.

Q4

Here's a step-by-step guide to performing the requested operations in Hive:

---

1. Create a Emplyee_Details Database

```
CREATE DATABASE IF NOT EXISTS Emplyee_Details;

USE Emplyee_Details;
```

---

2. Create a table called Employee

```
CREATE TABLE Employee (
   eid INT,
   ename STRING,
   dob DATE,
   hiredate DATE,
   designation STRING,
   salary FLOAT,
   commission FLOAT,
   dept STRING
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE;
```

---

3. Add 10 Employee Details from a .csv File

1. Save the 10 employee details in a CSV file (e.g., employee_data.csv).
2. Move the file to HDFS:
3. hdfs dfs -put /local/path/employee_data.csv /user/hive/warehouse/employee_data.csv
4. Load the data into the Employee table:
5. LOAD DATA INPATH '/user/hive/warehouse/employee_data.csv' INTO TABLE Employee;

---

4. Display the Department-wise Details

```
SELECT dept, COUNT(*) AS num_employees, AVG(salary) AS avg_salary
FROM Employee
GROUP BY dept;
```

---

5. Display the eid and ename of Employees Whose Salary > 45000

```
SELECT eid, ename
FROM Employee
WHERE salary > 45000;
```

---

6. Rename the Table to Employee_Details

```
ALTER TABLE Employee RENAME TO Employee_Details;
```

---

7. Rename the Column designation to job_title

```
ALTER TABLE Employee_Details CHANGE designation job_title STRING;
```

---

8. Create a Partitioned Table Based on Department

1. Create the new partitioned table:
2. CREATE TABLE Employee_Details_Partitioned (
3.   eid INT,
4.   ename STRING,
5.   dob DATE,
6.   hiredate DATE,
7.   job_title STRING,
8.   salary FLOAT,
9.   commission FLOAT
10.)
11. PARTITIONED BY (dept STRING)
12. ROW FORMAT DELIMITED
13. FIELDS TERMINATED BY ','

14. STORED AS TEXTFILE;
15. Enable dynamic partitioning:
16. SET hive.exec.dynamic.partition = true;
17. SET hive.exec.dynamic.partition.mode = nonstrict;
18. Insert data from the non-partitioned table into the partitioned table:
19. INSERT OVERWRITE TABLE Employee_Details_Partitioned
20. PARTITION (dept)
21. SELECT eid, ename, dob, hiredate, job_title, salary, commission, dept
22. FROM Employee_Details;

This workflow ensures all operations are carried out in Hive seamlessly. Let me know if you need further assistance!

Q5

Here's a step-by-step Pig script to handle the requested operations. Save it as a .pig file (e.g., employee_department_operations.pig) and execute it using the Pig command-line interface.

Step 1: Create emp.txt and dept.txt files

Create the files in your local file system with the following sample data:

emp.txt

1,John,Mumbai,101,50000

2,Alice,Pune,102,55000

3,Robert,Mumbai,103,45000

4,Maria,Bangalore,101,47000

5,James,Delhi,104,52000

6,Linda,Mumbai,102,60000

7,William,Delhi,105,53000

8,Sophia,Pune,103,48000

9,Liam,Mumbai,105,49000

10,Emma,Bangalore,104,55000

dept.txt

101,HR,Mumbai

102,Finance,Pune

103,IT,Bangalore

104,Marketing,Delhi

105,Sales,Chennai

Step 2: Load Data into HDFS

Move the files to HDFS:

hdfs dfs -put /local/path/emp.txt /user/hadoop/emp.txt

hdfs dfs -put /local/path/dept.txt /user/hadoop/dept.txt

Pig Script: employee_department_operations.pig

```
-- Load the Employee and Department data

Employee = LOAD '/user/hadoop/emp.txt' USING PigStorage(',')

      AS (eno:INT, name:CHARARRAY, city:CHARARRAY, did:INT, salary:FLOAT);


Department = LOAD '/user/hadoop/dept.txt' USING PigStorage(',')

       AS (did:INT, dname:CHARARRAY, location:CHARARRAY);


-- Step 3: Create relation Employee (already done during the load)


-- Step 4: Create a relation Department with 5 records (done during load, assuming only 5 records in dept.txt)


-- Step 5: Display all employees from city "Mumbai"

MumbaiEmployees = FILTER Employee BY city == 'Mumbai';

DUMP MumbaiEmployees;


-- Step 6: Display dname of each employee (join Employee with Department)

EmployeeWithDept = JOIN Employee BY did, Department BY did;

EmployeeDName = FOREACH EmployeeWithDept GENERATE Employee::eno, Employee::name, Department::dname;
```

```
DUMP EmployeeDName;


-- Step 7: Display the department-wise employee count

DepartmentCount = GROUP Employee BY did;

DepartmentWiseCount = FOREACH DepartmentCount GENERATE group AS did, COUNT(Employee) AS emp_count;

DUMP DepartmentWiseCount;


-- Step 8: Perform inner join between Employee and Department

InnerJoinResult = JOIN Employee BY did, Department BY did;

DUMP InnerJoinResult;


-- Step 9: Sort the employee details according to their name in descending order

SortedEmployees = ORDER Employee BY name DESC;

DUMP SortedEmployees;
```

Step 3: Execute the Script

Run the script using the Pig command:

```
pig -x mapreduce employee_department_operations.pig
```

Explanation of Steps:

1. **Load Data**: The LOAD command reads the emp.txt and dept.txt files.
2. **Filtering**: Use FILTER to retrieve employees from Mumbai.
3. **Join Operations**: Use JOIN to combine the Employee and Department relations.
4. **Grouping**: Use GROUP to group employees by department and count them.
5. **Sorting**: Use ORDER to sort employee details in descending order by name.

This script performs all the operations requested and uses efficient Pig commands for processing the data. Let me know if you need further clarifications!

Q6 Perform the following operations in MongoDB:

1. Create a Customer_Details Database

2. Create a collection called ‘Customer’

3. Add 10 customer in the ‘Customer’ database. Give each document the following

properties:

cust_code,cust_name,cust_city,working_area,phone_no,agent_code

4. Display all information of customer where Names begin with ‘T’ or ‘S’

5. Update collection to modify customer ‘Rahul Pandey’ cust_city to ‘Mumbai’

6. Delete all documents where customer have agent code as AG002

7. Display the cust_code and cust_name of all customers whose city is “Delhi”

Here’s a step-by-step guide to performing the requested operations in MongoDB:

---

1. Create a Customer_Details Database

use Customer_Details;

---

2. Create a Collection Called Customer

MongoDB automatically creates a collection when you insert a document, so there's no need for an explicit creation step. However, you can manually create an empty collection:

db.createCollection("Customer");

---

3. Add 10 Customers to the Customer Collection

Insert 10 sample customer records:

db.Customer.insertMany([

{ cust_code: "C001", cust_name: "Rahul Pandey", cust_city: "Delhi", working_area: "Connaught Place", phone_no: "9876543210", agent_code: "AG001" },

{ cust_code: "C002", cust_name: "Sunil Kumar", cust_city: "Mumbai", working_area: "Andheri", phone_no: "9876543211", agent_code: "AG002" },

{ cust_code: "C003", cust_name: "Sita Sharma", cust_city: "Delhi", working_area: "Karol Bagh", phone_no: "9876543212", agent_code: "AG003" },

{ cust_code: "C004", cust_name: "Tina Kapoor", cust_city: "Pune", working_area: "Koregaon Park", phone_no: "9876543213", agent_code: "AG001" },

{ cust_code: "C005", cust_name: "Amit Gupta", cust_city: "Kolkata", working_area: "Salt Lake", phone_no: "9876543214", agent_code: "AG002" },

```
    { cust_code: "C006", cust_name: "Simran Kaur", cust_city: "Chennai", working_area: "T. Nagar", phone_no: "9876543215", agent_code: "AG003" },

    { cust_code: "C007", cust_name: "Ravi Verma", cust_city: "Delhi", working_area: "Rohini", phone_no: "9876543216", agent_code: "AG002" },

    { cust_code: "C008", cust_name: "Sonal Singh", cust_city: "Bangalore", working_area: "Whitefield", phone_no: "9876543217", agent_code: "AG001" },

    { cust_code: "C009", cust_name: "Tarun Joshi", cust_city: "Hyderabad", working_area: "Banjara Hills", phone_no: "9876543218", agent_code: "AG003" },

    { cust_code: "C010", cust_name: "Pooja Mishra", cust_city: "Mumbai", working_area: "Malad", phone_no: "9876543219", agent_code: "AG002" }

]);
```

---

4. Display All Information of Customers Where Names Begin with 'T' or 'S'

```
db.Customer.find({ cust_name: { $regex: "^[TS]" } });
```

---

5. Update Customer Rahul Pandey's cust_city to Mumbai

```
db.Customer.updateOne({ cust_name: "Rahul Pandey" }, { $set: { cust_city: "Mumbai" } });
```

---

6. Delete All Documents Where Customers Have Agent Code as AG002

```
db.Customer.deleteMany({ agent_code: "AG002" });
```

---

7. Display the cust_code and cust_name of All Customers Whose City is Delhi

```
db.Customer.find({ cust_city: "Delhi" }, { _id: 0, cust_code: 1, cust_name: 1 });
```

Explanation:

1. **Insert Data**: Use insertMany for batch insertion of multiple records.
2. **Query with Regex**: $regex is used for pattern matching to filter names starting with T or S.
3. **Update Record**: updateOne modifies a single document matching the condition.
4. **Delete Records**: deleteMany removes multiple documents based on a condition.
5. **Projection**: Specify fields in the second argument of find to control the displayed output.

This script can be executed in the MongoDB shell or a GUI client like MongoDB Compass. Let me know if you need further assistance!

Here’s a Python program using the pymongo library to perform the described MongoDB operations:

Prerequisites

1. Install pymongo library:
2. pip install pymongo
3. Ensure MongoDB is running on your system.

Python Script

```
from pymongo import MongoClient, ASCENDING, DESCENDING

# i. Start MongoDB connection
client = MongoClient("mongodb://localhost:27017/")
print("Connected to MongoDB!")

# ii. Create database MCA2022
db = client["MCA2022"]
print("Database MCA2022 created!")

# iii. Create collections Employee and Customer
employee_col = db["Employee"]
customer_col = db["Customer"]
print("Collections Employee and Customer created!")

# iv. Insert documents (single and multiple)
# Insert a single employee
employee_col.insert_one({
  "emp_id": 1,
  "emp_fname": "John",
  "emp_lname": "Doe",
  "dept": "IT",
  "sal": 75000
})
# Insert multiple employees
employee_col.insert_many([
    {"emp_id": 2, "emp_fname": "Jane", "emp_lname": "Smith", "dept": "HR", "sal": 65000},
    {"emp_id": 3, "emp_fname": "Mike", "emp_lname": "Johnson", "dept": "Finance", "sal": 80000},
    {"emp_id": 4, "emp_fname": "Emily", "emp_lname": "Davis", "dept": "IT", "sal": 70000}
])

# Insert a single customer
customer_col.insert_one({
    "cust_id": 1,
    "cust_fname": "Alice",
    "cust_lname": "Brown",
    "contact_no": "9876543210",
    "address": "123 Main St"
})
# Insert multiple customers
customer_col.insert_many([
    {"cust_id": 2, "cust_fname": "Bob", "cust_lname": "White", "contact_no": "9876543211",
"address": "456 Elm St"},
    {"cust_id": 3, "cust_fname": "Charlie", "cust_lname": "Green", "contact_no": "9876543212",
"address": "789 Pine St"}
])

print("Documents inserted successfully!")

# v. Find employee according to emp_id
emp_id = 1
employee = employee_col.find_one({"emp_id": emp_id})
print(f"Employee with emp_id {emp_id}: {employee}")

# vi. Find customer according to cust_id
```

```
cust_id = 1
customer = customer_col.find_one({"cust_id": cust_id})
print(f"Customer with cust_id {cust_id}: {customer}")

# vii. Update employee salary
new_salary = 85000
employee_col.update_one({"emp_id": emp_id}, {"$set": {"sal": new_salary}})
print(f"Updated salary of employee with emp_id {emp_id} to {new_salary}!")

# viii. Delete a document
employee_col.delete_one({"emp_id": 4})
print("Deleted employee with emp_id 4!")

# ix. Create an index on employee
employee_col.create_index([("sal", ASCENDING)])
print("Index created on employee salary!")

# x. Get list of indexes
indexes = employee_col.index_information()
print(f"List of indexes: {indexes}")

# xi. Drop the index
employee_col.drop_index("sal_1")
print("Index on salary dropped!")

# xii. Sort employees by salary in descending order
sorted_employees = employee_col.find().sort("sal", DESCENDING)
print("Employees sorted by salary in descending order:")
for emp in sorted_employees:
    print(emp)
```

Explanation of the Operations

1. **Start MongoDB**: The script connects to the local MongoDB server using MongoClient.
2. **Create Database and Collections**: Use client["MCA2022"] to create a database and db["CollectionName"] for collections.
3. **Insert Documents**: Use insert_one for a single document and insert_many for multiple.
4. **Find Documents**: Use find_one with a query filter to fetch a specific document.
5. **Update Document**: Use update_one with $set to update specific fields.
6. **Delete Document**: Use delete_one to remove a document.
7. **Create Index**: Use create_index to create an index, here on the sal field.
8. **Get Indexes**: Use index_information to list all indexes in a collection.
9. **Drop Index**: Use drop_index with the index name to remove it.
10. **Sort Data**: Use find().sort(field, direction) to sort documents.

Output Example

```
Connected to MongoDB!

Database MCA2022 created!

Collections Employee and Customer created!

Documents inserted successfully!

Employee with emp_id 1: {'_id': ObjectId(...), 'emp_id': 1, 'emp_fname': 'John', 'emp_lname': 'Doe', 'dept': 'IT', 'sal': 75000}

Customer with cust_id 1: {'_id': ObjectId(...), 'cust_id': 1, 'cust_fname': 'Alice', 'cust_lname': 'Brown', 'contact_no': '9876543210', 'address': '123 Main St'}

Updated salary of employee with emp_id 1 to 85000!

Deleted employee with emp_id 4!

Index created on employee salary!

List of indexes: {'_id_': {'key': [('_id', 1)]}, 'sal_1': {'key': [('sal', 1)]}}

Index on salary dropped!

Employees sorted by salary in descending order:

{'_id': ObjectId(...), 'emp_id': 3, 'emp_fname': 'Mike', 'emp_lname': 'Johnson', 'dept': 'Finance', 'sal': 80000}

{'_id': ObjectId(...), 'emp_id': 1, 'emp_fname': 'John', 'emp_lname': 'Doe', 'dept': 'IT', 'sal': 85000}

...
```

This script provides a complete solution for managing Employee and Customer collections with MongoDB. Let me know if you need further assistance!