**1.Implement program on wordcount.**

import java.io.IOException; import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration; import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.IntWritable; import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Job; import org.apache.hadoop.mapreduce.Mapper; import org.apache.hadoop.mapreduce.Reducer;

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

public class WordCount {

public static class TokenizerMapper

extends Mapper<Object, Text, Text, IntWritable>{

private final static IntWritable one = new IntWritable(1); private Text word = new Text();

public void map(Object key, Text value, Context context

) throws IOException, InterruptedException { StringTokenizer itr = new StringTokenizer(value.toString()); while (itr.hasMoreTokens()) {

word.set(itr.nextToken()); context.write(word, one);

}

}

}

public static class IntSumReducer

extends Reducer<Text,IntWritable,Text,IntWritable> { private IntWritable result = new IntWritable();

public void reduce(Text key, Iterable<IntWritable> values,

Context context

) throws IOException, InterruptedException {

int sum = 0;

for (IntWritable val : values) { sum += val.get();

}

result.set(sum); context.write(key, result);

}

}

public static void main(String[] args) throws Exception { Configuration conf = new Configuration();

Job job = Job.getInstance(conf, "word count"); job.setJarByClass(WordCount.class); job.setMapperClass(TokenizerMapper.class); job.setCombinerClass(IntSumReducer.class); job.setReducerClass(IntSumReducer.class); job.setOutputKeyClass(Text.class); job.setOutputValueClass(IntWritable.class); FileInputFormat.addInputPath(job, new Path(args[0]));

FileOutputFormat.setOutputPath(job, new Path(args[1])); System.exit(job.waitForCompletion(true) ? 0 : 1);

}

}

## EXPERIMENT 2

**2. Write a program to read, write operations from local file system to HDFS using File System API.**

## Program for get command:

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

public class Download

{

public static void main(String s[])

{

Configuration conf=new Configuration(); FileSystem fs;

try

{

fs=FileSystem.get(conf); Path src=new Path(s[0]); Path dst=new Path(s[1]); fs.copyToLocalFile(src, dst);

}

catch(Exception e)

{

System.out.println(e);

}

}

}

## Program for put command:

import org.apache.hadoop.conf.\*; import org.apache.hadoop.fs.\*; public class Upload

{

public static void main(String s[])

{

Configuration conf=new Configuration(); FileSystem fs;

try

{

fs=FileSystem.get(conf);

Path src=new Path(s[0]); Path dst=new Path(s[1]);

fs.copyFromLocalFile(src, dst); fs.close();

}

catch(Exception e)

{

System.out.println(e);

}

}

}

## Program for read operation:

import java.io.DataInputStream; import org.apache.hadoop.conf.\*; import org.apache.hadoop.fs.\*;

public class ReadFile

{

public static void main(String s[])

{

String filename;

String data = null;

DataInputStream ds=new DataInputStream(System.in);

Configuration conf=new Configuration(); FSDataInputStream fsi;

FileSystem fs; try

{

System.out.println("Enter the file to be displayed"); filename=ds.readLine();

fs=FileSystem.get(conf); fsi=fs.open(new Path(filename)); System.out.println("Reading ");

while((data=fsi.readLine())!= null)

{

System.out.println("in progress. ");

System.out.println(data);

}

}

catch(Exception e)

{

System.out.println(e);

}

}

}

**3. Implement Map-Reduce application to find sum of salaries of employees for each department in java** INPUT FILE

emp1 dept1 5000

emp2 dept1 6000

emp3 dept2 4500

emp4 dept2 5500

emp5 dept3 7000

PROGRAM

import java.io.IOException;

import org.apache.hadoop.conf.Configuration; import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.IntWritable; import org.apache.hadoop.io.LongWritable; import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Job; import org.apache.hadoop.mapreduce.Mapper; import org.apache.hadoop.mapreduce.Reducer;

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

public class EmployeeSalary {

// Mapper Class

public static class SalaryMapper extends Mapper<LongWritable, Text, Text, IntWritable> { private Text department = new Text();

private IntWritable salary = new IntWritable();

@Override

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

String line = value.toString(); String[] fields = line.split("\\s+"); if (fields.length == 3) {

department.set(fields[1]); salary.set(Integer.parseInt(fields[2])); context.write(department, salary);

}

}

}

// Reducer Class

public static class SalaryReducer extends Reducer<Text, IntWritable, Text, IntWritable> { @Override

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

int sum = 0;

for (IntWritable val : values) { sum += val.get();

}

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

}

}

// Driver Class

public static void main(String[] args) throws Exception { if (args.length != 2) {

System.err.println("Usage: EmployeeSalary <input path> <output path>"); System.exit(-1);

}

Configuration conf = new Configuration();

Job job = Job.getInstance(conf, "Employee Salary Sum");

job.setJarByClass(EmployeeSalary.class); job.setMapperClass(SalaryMapper.class); job.setReducerClass(SalaryReducer.class);

job.setOutputKeyClass(Text.class); job.setOutputValueClass(IntWritable.class);

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

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

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

}

}

### 4. Write a pig script for log file analysis.

**Step-1:** Create a text file with Name stud and the write the fields will be rollno, name, age and city. for example:

stud.txt

11,aa,21,hyderabad 22,bb,23,delhi 33,cc,22,chennai 44,dd,22,mumbai 55,ee,23,delhi

**Step-2:** Create a file with .pig and write the commands in it p2.pig:

s2 = Load '/home/cloudera/stud' USING PigStorage(',') as (id:int,name:chararray,age:int,city:chararray);

DUMP s2;

p3.pig:

s2 = Load '/home/cloudera/stud' USING PigStorage(',') as (id:int,name:chararray,age:int,city:chararray);

DUMP s2;

sgroup = GROUP s2 ALL; DUMP sgroup;

savg = FOREACH sgroup GENERATE (s2.name,s2.id), AVG(s2.age); DUMP savg;

p4.pig:

s2 = Load '/home/cloudera/stud' USING PigStorage(',') as (id:int,name:chararray,age:int,city:chararray);

DUMP s2;

supper = FOREACH s2 GENERATE (id,name), UPPER(name); DUMP supper;

p4.pig:

s2 = Load '/home/cloudera/stud' USING PigStorage(',') as (id:int,name:chararray,age:int,city:chararray);

DUMP s2;

smath = FOREACH s2 GENERATE (id,name), SQRT(name); DUMP smath;

**Step-3:-** Open the terminal and execute the program example: for executing p2.pig file

grunt> pig -x local p2.pig

with this we can execute the pig commands which are written in the file.

1. Implement a pig script to analyze the data using UDFs.

To analyze data using Pig (a high-level platform for creating programs that run on Apache Hadoop), you'll need to create a Pig script and User Defined Functions (UDFs) to perform specific tasks. Below is an example of how to do this:

# Example Data

Let's assume we have a dataset called data.txt with the following format:

1,John,2000

2,Jane,3000

3,Bob,1500

4,Alice,2500 5,Tom,1200

The dataset represents records with the following fields: id, name, and salary.

# Pig Script

### Load the data

* 1. **Register the UDF**

### Use the UDF to analyze the data

***Step 1: Write the UDF***

We'll write a simple UDF in Java to calculate a bonus based on salary. For example, let's give a 10% bonus to each salary.

Create a Java file BonusCalculator.java:

import java.io.IOException; import org.apache.pig.EvalFunc; import org.apache.pig.data.Tuple;

public class BonusCalculator extends EvalFunc<Double> {

@Override

public Double exec(Tuple input) throws IOException { if (input == null || input.size() == 0) {

return null;

}

try {

Double salary = (Double) input.get(0); return salary \* 0.10;

} catch (Exception e) {

throw new IOException("Caught exception processing input row ", e);

}

}

}

Compile the Java code and package it into a JAR file.

***Step 2: Write the Pig Script***

Create a Pig script analyze\_data.pig:

-- Load the data

data = LOAD 'data.txt' USING PigStorage(',') AS (id:int, name:chararray, salary:double);

-- Register the UDF

REGISTER 'path/to/your/udf.jar';

-- Define the UDF

DEFINE BonusCalculator BonusCalculator();

-- Calculate the bonus

data\_with\_bonus = FOREACH data GENERATE id, name, salary, BonusCalculator(salary) AS bonus;

-- Store the result

STORE data\_with\_bonus INTO 'output' USING PigStorage(',');

# Steps to Execute

1. **Compile and package the UDF** javac -cp /path/to/pig.jar BonusCalculator.java jar -cvf udf.jar BonusCalculator.class

Run the Pig script pig analyze\_data.pig

# Explanation

1. **Loading Data**: The LOAD function reads data from data.txt.
2. **Registering UDF**: The REGISTER command makes the UDF available to the Pig script.
3. **Defining UDF**: DEFINE assigns a name to the UDF.
4. **Using UDF**: The FOREACH statement applies the BonusCalculator UDF to each record's salary to calculate the bonus.
5. **Storing Results**: The STORE function writes the result to an output directory.
6. Create Managed table and load a csv file from local storage. Verify the location of table data in HDFS and perform query operations. Delete the table and verify the data in HDFS.

To create a managed table in Hive, load a CSV file from local storage, verify the location of the table data in HDFS, perform query operations, and then delete the table and verify the data in HDFS, follow these steps:

# Step 1: Create and Load the Managed Table

## Start Hive:

hive

### Create the Managed Table:

CREATE TABLE employee ( id INT,

name STRING, salary DOUBLE

)

ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' STORED AS TEXTFILE;

**Load Data from Local Storage**: Assuming your CSV file data.csv is in the local filesystem, load it into the Hive table:

LOAD DATA LOCAL INPATH '/path/to/data.csv' INTO TABLE employee;

# Step 2: Verify the Location of Table Data in HDFS

1. **Check the HDFS Path**: Managed tables are stored in the default location, which is usually /user/hive/warehouse/<table\_name>. You can check this using the HDFS command:

hdfs dfs -ls /user/hive/warehouse/employee

# Step 3: Perform Query Operations

## 1. Run Some Queries

-- Select all records

SELECT \* FROM employee;

-- Count the number of records SELECT COUNT(\*) FROM employee;

-- Calculate the total salary

SELECT SUM(salary) FROM employee;

-- Find the highest salary

SELECT MAX(salary) FROM employee;

-- Select all records

SELECT \* FROM employee;

-- Count the number of records SELECT COUNT(\*) FROM employee;

-- Calculate the total salary

SELECT SUM(salary) FROM employee;

-- Find the highest salary

SELECT MAX(salary) FROM employee;

# Step 4: Delete the Table and Verify the Data in HDFS

1. **Drop the Table**:

DROP TABLE employee;

**Verify the Data in HDFS**: After dropping the table, the data should also be deleted from HDFS. hdfs dfs -ls /user/hive/warehouse/employee

# Complete Commands

Here's the complete sequence of commands to run in the Hive terminal and the shell:

**Hive Terminal:**

-- Create the table CREATE TABLE employee (

id INT,

name STRING, salary DOUBLE

)

ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' STORED AS TEXTFILE;

-- Load data from local storage

LOAD DATA LOCAL INPATH '/path/to/data.csv' INTO TABLE employee;

-- Run queries

SELECT \* FROM employee;

SELECT COUNT(\*) FROM employee; SELECT SUM(salary) FROM employee; SELECT MAX(salary) FROM employee;

-- Drop the table

DROP TABLE employee;

# Verify the HDFS path before dropping the table hdfs dfs -ls /user/hive/warehouse/employee

# Verify the HDFS path after dropping the table hdfs dfs -ls /user/hive/warehouse/employee

Replace /path/to/data.csv with the actual path to your CSV file. This sequence will create a managed table, load data, run queries, drop the table, and verify the data's presence in HDFS at each step.

1. Create External table and load a csv file from local storage. The data should be stored in ORC format in specified folder in HDFS. Perform query operations and verify the data in HDFS. Delete the table and verify the data in HDFS.

To create an external table in Hive, load a CSV file from local storage, store the data in ORC format in a specified folder in HDFS, perform query operations, and then delete the table while ensuring the data remains in HDFS, follow these steps:

# Step 1: Prepare HDFS Directory

## Create an HDFS Directory:

hdfs dfs -mkdir -p /user/hive/external/employee\_orc

# Step 2: Create and Load the External Table

1. **Start Hive**:

hive

Create the External Table

CREATE EXTERNAL TABLE employee\_external ( id INT,

name STRING, salary DOUBLE

)

ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' STORED AS TEXTFILE

LOCATION '/user/hive/external/employee\_orc';

**Load Data from Local Storage**: Assuming your CSV file data.csv is in the local filesystem, load it into the Hive table:

LOAD DATA LOCAL INPATH '/path/to/data.csv' INTO TABLE employee\_external;

# Step 3: Convert Data to ORC Format

## 1. Create a Table in ORC Format:

CREATE TABLE employee\_orc ( id INT,

name STRING, salary DOUBLE

)

STORED AS ORC;

### Insert Data into ORC Table:

INSERT INTO TABLE employee\_orc SELECT \* FROM employee\_external;

### Move Data to Specified HDFS Folder:

INSERT OVERWRITE DIRECTORY '/user/hive/external/employee\_orc' STORED AS ORC

SELECT \* FROM employee\_orc;

# Step 4: Verify the Data in HDFS

## 1. Check the HDFS Path

hdfs dfs -ls /user/hive/external/employee\_orc

# Step 5: Perform Query Operations

## 1. Run Some Queries on the ORC Table

-- Select all records

SELECT \* FROM employee\_orc;

-- Count the number of records

SELECT COUNT(\*) FROM employee\_orc;

-- Calculate the total salary

SELECT SUM(salary) FROM employee\_orc;

-- Find the highest salary

SELECT MAX(salary) FROM employee\_orc;

# Step 6: Delete the Table and Verify the Data in HDFS

## 1. Drop the External Table:

DROP TABLE employee\_external;

**Verify the Data in HDFS**: The data should still be in HDFS because it was loaded as an external table.

hdfs dfs -ls /user/hive/external/employee\_orc

# Complete Commands

Here's the complete sequence of commands to run in the Hive terminal and the shell:

**Hive Terminal:**

-- Create the external table

CREATE EXTERNAL TABLE employee\_external ( id INT,

name STRING, salary DOUBLE

)

ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' STORED AS TEXTFILE

LOCATION '/user/hive/external/employee\_orc';

-- Load data from local storage

LOAD DATA LOCAL INPATH '/path/to/data.csv' INTO TABLE employee\_external;

-- Create the ORC table CREATE TABLE employee\_orc (

id INT,

name STRING, salary DOUBLE

)

STORED AS ORC;

-- Insert data into the ORC table INSERT INTO TABLE employee\_orc SELECT \* FROM employee\_external;

-- Move data to the specified HDFS folder in ORC format

INSERT OVERWRITE DIRECTORY '/user/hive/external/employee\_orc' STORED AS ORC

SELECT \* FROM employee\_orc;

-- Run queries

SELECT \* FROM employee\_orc;

SELECT COUNT(\*) FROM employee\_orc; SELECT SUM(salary) FROM employee\_orc; SELECT MAX(salary) FROM employee\_orc;

-- Drop the external table

DROP TABLE employee\_external;

Shell:

# Create the HDFS directory

hdfs dfs -mkdir -p /user/hive/external/employee\_orc

# Verify the HDFS path before dropping the table hdfs dfs -ls /user/hive/external/employee\_orc

# Verify the HDFS path after dropping the table hdfs dfs -ls /user/hive/external/employee\_orc

Replace /path/to/data.csv with the actual path to your CSV file. This sequence will create an external table, load data, convert it to ORC format, run queries, drop the table, and verify the data's presence in HDFS at each step.

1. Create partitioned table using Static partitioning technique load the data into corresponding partitions.

To create a partitioned table using static partitioning in Hive and load data into corresponding partitions, follow these steps:

# Step 1: Prepare the Data

Assume we have a CSV file data.csv with the following content:

1,John,2000,2023

2,Jane,3000,2023

3,Bob,1500,2022

4,Alice,2500,2022

5,Tom,1200,2021

The dataset represents records with the following fields: id, name, salary, and year.

# Step 2: Create the Partitioned Table

## Start Hive:

hive

### Create the Partitioned Table:

CREATE TABLE employee\_partitioned ( id INT,

name STRING, salary DOUBLE

)

PARTITIONED BY (year INT) ROW FORMAT DELIMITED

FIELDS TERMINATED BY ',' STORED AS TEXTFILE;

# Step 3: Load Data into Corresponding Partitions

Since we're using static partitioning, we need to load data into each partition separately. First, we should split the data.csv file into separate files for each partition.

1. **Split Data into Separate Files**:

o data\_2023.csv:

1,John,2000

2,Jane,3000

data\_2022.csv:

3,Bob,1500

4,Alice,2500

data\_2021.csv:

5,Tom,1200

### Load Data into Partitions:

-- Load data for the year 2023

LOAD DATA LOCAL INPATH '/path/to/data\_2023.csv' INTO TABLE employee\_partitioned PARTITION (year=2023);

-- Load data for the year 2022

LOAD DATA LOCAL INPATH '/path/to/data\_2022.csv' INTO TABLE employee\_partitioned PARTITION (year=2022);

-- Load data for the year 2021

LOAD DATA LOCAL INPATH '/path/to/data\_2021.csv' INTO TABLE employee\_partitioned PARTITION (year=2021);

# Step 4: Verify the Data

## 1. Run Queries to Verify Data:

-- Select all records for the year 2023

SELECT \* FROM employee\_partitioned WHERE year=2023;

-- Select all records for the year 2022

SELECT \* FROM employee\_partitioned WHERE year=2022;

-- Select all records for the year 2021

SELECT \* FROM employee\_partitioned WHERE year=2021;

-- Select all records from the partitioned table SELECT \* FROM employee\_partitioned;

# Complete Commands

Here's the complete sequence of commands to run in the Hive terminal and the shell:

**Hive Terminal:**

-- Create the partitioned table

CREATE TABLE employee\_partitioned ( id INT,

name STRING, salary DOUBLE

)

PARTITIONED BY (year INT) ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' STORED AS TEXTFILE;

-- Load data into corresponding partitions

LOAD DATA LOCAL INPATH '/path/to/data\_2023.csv' INTO TABLE employee\_partitioned PARTITION (year=2023);

LOAD DATA LOCAL INPATH '/path/to/data\_2022.csv' INTO TABLE employee\_partitioned PARTITION (year=2022);

LOAD DATA LOCAL INPATH '/path/to/data\_2021.csv' INTO TABLE employee\_partitioned PARTITION (year=2021);

-- Run queries to verify data

SELECT \* FROM employee\_partitioned WHERE year=2023; SELECT \* FROM employee\_partitioned WHERE year=2022; SELECT \* FROM employee\_partitioned WHERE year=2021; SELECT \* FROM employee\_partitioned;

Replace /path/to/data\_2023.csv, /path/to/data\_2022.csv, and /path/to/data\_2021.csv with the actual paths to your split CSV files. This sequence will create a partitioned table, load data into the respective partitions, and verify the data through queries.

10. Create and manage a table with bucketing concept and execute queries with UDFs.

To create and manage a table with the bucketing concept in Hive and execute queries with User Defined Functions (UDFs), follow these steps:

# Step 1: Prepare the Data

Assume we have a CSV file data.csv with the following content:

1,John,2000

2,Jane,3000

3,Bob,1500

4,Alice,2500 5,Tom,1200

The dataset represents records with the following fields: id, name, and salary.

# Step 2: Create and Load a Bucketed Table

1. **Start Hive**: hive

### Create a Bucketed Table:

CREATE TABLE employee\_bucketed ( id INT,

name STRING, salary DOUBLE

)

CLUSTERED BY (id) INTO 4 BUCKETS ROW FORMAT DELIMITED

FIELDS TERMINATED BY ',' STORED AS TEXTFILE;

### Load Data into the Bucketed Table:

-- Set the following properties to enable dynamic bucketing

SET hive.enforce.bucketing=true;

-- Load data from local storage

LOAD DATA LOCAL INPATH '/path/to/data.csv' INTO TABLE employee\_bucketed;

# Step 3: Verify the Bucketing

## 1. Verify Data Distribution Across Buckets

-- Query to count the number of records in each bucket SELECT \*, INPUT FILE NAME FROM employee\_bucketed;

# Step 4: Create and Use UDFs

1. **Write a Simple UDF**: Let's create a UDF in Java to calculate a bonus based on salary. For example, we'll give a 10% bonus to each salary.

Create a Java file BonusCalculator.java:

import java.io.IOException; import org.apache.pig.EvalFunc; import org.apache.pig.data.Tuple;

public class BonusCalculator extends EvalFunc<Double> { @Override

public Double exec(Tuple input) throws IOException { if (input == null || input.size() == 0) {

return null;

}

try {

Double salary = (Double) input.get(0); return salary \* 0.10;

} catch (Exception e) {

throw new IOException("Caught exception processing input row ", e);

}

}

}

Compile the Java code and package it into a JAR file:

javac -cp /path/to/hive/lib/\*:. BonusCalculator.java jar -cvf BonusCalculator.jar BonusCalculator.class

### Register the UDF in Hive:

ADD JAR /path/to/BonusCalculator.jar;

CREATE TEMPORARY FUNCTION bonus\_calculator AS 'BonusCalculator';

### Use the UDF in Queries:

-- Select all records with the calculated bonus

SELECT id, name, salary, bonus\_calculator(salary) AS bonus FROM employee\_bucketed;

# Complete Commands

Here's the complete sequence of commands to run in the Hive terminal and the shell:

**Hive Terminal:**

-- Create the bucketed table

CREATE TABLE employee\_bucketed ( id INT,

name STRING, salary DOUBLE

)

CLUSTERED BY (id) INTO 4 BUCKETS ROW FORMAT DELIMITED

FIELDS TERMINATED BY ',' STORED AS TEXTFILE;

-- Enable dynamic bucketing

SET hive.enforce.bucketing=true;

-- Load data into the bucketed table

LOAD DATA LOCAL INPATH '/path/to/data.csv' INTO TABLE employee\_bucketed;

-- Verify data distribution across buckets

SELECT \*, INPUT FILE NAME FROM employee\_bucketed;

-- Register the UDF

ADD JAR /path/to/BonusCalculator.jar;

CREATE TEMPORARY FUNCTION bonus\_calculator AS 'BonusCalculator';

-- Use the UDF in queries

SELECT id, name, salary, bonus\_calculator(salary) AS bonus FROM employee\_bucketed;

Shell:

# Compile the Java UDF

javac -cp /path/to/hive/lib/\*:. BonusCalculator.java

# Create a JAR file for the UDF

jar -cvf BonusCalculator.jar BonusCalculator.class

Replace /path/to/data.csv with the actual path to your CSV file and

/path/to/BonusCalculator.jar with the actual path to your JAR file. This sequence will create a bucketed table, load data into it, register and use a UDF, and execute queries with the UDF.

11. Implement flume script for spooling a directory from local file system to HDFS.

To implement a Flume script for spooling a directory from the local file system to HDFS, follow these steps:

# Step 1: Install and Configure Apache Flume

## Download Apache Flume:

* + Download the Flume binary from the [Apache Flume downloads page.](https://flume.apache.org/download.html)
  + Extract the downloaded archive.

## Set Up Environment Variables:

* + Add the Flume bin directory to your PATH.
  + Example (in ~/.bashrc or ~/.bash\_profile

export FLUME\_HOME=/path/to/flume export PATH=$PATH:$FLUME\_HOME/bin

## Create Directories:

* + Create a directory for the files to be spooled, e.g., /path/to/spooldir.
  + Ensure the HDFS directory you want to write to exists, e.g.,

/user/flume/spooldir.

# Step 2: Create the Flume Configuration File

Create a Flume configuration file, e.g., flume-conf.properties, with the following content:

# Name the components on this agent agent1.sources = spool-source agent1.sinks = hdfs-sink agent1.channels = memory-channel

# Describe/configure the source agent1.sources.spool-source.type = spooldir

agent1.sources.spool-source.spoolDir = /path/to/spooldir agent1.sources.spool-source.fileHeader = true

# Describe the sink agent1.sinks.hdfs-sink.type = hdfs

agent1.sinks.hdfs-sink.hdfs.path = hdfs://namenode:8020/user/flume/spooldir agent1.sinks.hdfs-sink.hdfs.fileType = DataStream

agent1.sinks.hdfs-sink.hdfs.writeFormat = Text agent1.sinks.hdfs-sink.hdfs.batchSize = 1000

agent1.sinks.hdfs-sink.hdfs.rollSize = 0

agent1.sinks.hdfs-sink.hdfs.rollCount = 10000

agent1.sinks.hdfs-sink.hdfs.rollInterval = 600

# Use a memory channel agent1.channels.memory-channel.type = memory

agent1.channels.memory-channel.capacity = 10000

agent1.channels.memory-channel.transactionCapacity = 1000

# Bind the source and sink to the channel agent1.sources.spool-source.channels = memory-channel agent1.sinks.hdfs-sink.channel = memory-channel

# Step 3: Start the Flume Agent

Run the Flume agent with the configuration file:

flume-ng agent --conf $FLUME\_HOME/conf --conf-file /path/to/flume-conf.properties --name agent1 - Dflume.root.logger=INFO,console

Replace /path/to/flume-conf.properties with the path to your configuration file.

# Step 4: Verify Data in HDFS

1. **Check HDFS Directory**: Ensure the data is being written to the HDFS directory specified in the configuration file:

hdfs dfs -ls /user/flume/spooldir

This configuration sets up a Flume agent to monitor a local directory for new files and write the contents of these files to HDFS. The memory channel serves as an intermediary buffer. Adjust the paths and parameters as needed for your environment.