Godavari Foundation's

Godavari Institute Of Management & Research, Jalgaon

Name: _ Roll No: _ _ _ _ _

Date of Performance: _ _ /_ _/20_ _ Batch: _ _ _ _ _ _

Class: M.C.A. (I) Practical no: **1**

Subject: 440(B) Lab on Big Data Analytics

Title: Use Hadoop File System (HDFS) commands to perform basic operations like creating directories, uploading files, listing files, and deleting files in HDFS.

- **Hadoop Installation:** Ensure you have Hadoop installed and configured correctly.
- **Hadoop Daemons Running:** Start the HDFS daemons (NameNode and DataNode) using the `start-dfs.sh` script (usually found in the `sbin` directory of your Hadoop installation).

HDFS Commands and Steps:

1. **Creating a Directory:**
    - Command: `hdfs dfs -mkdir <directory_path>`
    - Explanation: This command creates a new directory in HDFS.
    - Steps:
        - Open your terminal or command prompt.
        - Type: `hdfs dfs -mkdir /my_hdfs_directory` (Replace `/my_hdfs_directory` with the desired path).
        - Press Enter.
2. **Uploading a File from Local File System to HDFS:**
    - Command: `hdfs dfs -put <local_file_path> <hdfs_destination_path>`
    - Explanation: This command copies a file from your local file system to HDFS.
    - Steps:
        - Open your terminal.
        - Type: `hdfs dfs -put /path/to/local/file.txt /my_hdfs_directory/file.txt` (Replace `/path/to/local/file.txt` with the actual path to your local file and `/my_hdfs_directory/file.txt` with the desired HDFS destination).
        - Press Enter.
3. **Listing Files and Directories in HDFS:**
    - Command: `hdfs dfs -ls <hdfs_path>`
    - Explanation: This command lists the files and directories present at the specified path in HDFS.
    - Steps:
        - Open your terminal.
        - To list the contents of the root directory: `hdfs dfs -ls /`

        - To list the contents of the directory created previously: `hdfs dfs -ls /my_hdfs_directory`
        - Press Enter.

4. **Deleting a File or Directory in HDFS:**
    - Command: `hdfs dfs -rm <hdfs_file_path>` (for files)
    - Command: `hdfs dfs -rm -r <hdfs_directory_path>` (for directories, `-r` for recursive deletion)
    - Explanation: These commands remove files or directories from HDFS. Use `-r` to delete directories and their contents.
    - Steps:
        - Open your terminal.
        - To delete a file: `hdfs dfs -rm /my_hdfs_directory/file.txt`
        - To delete a directory and its content: `hdfs dfs -rm -r /my_hdfs_directory`
        - Press Enter.
5. **Viewing the content of a file:**
    - Command: `hdfs dfs -cat <hdfs_file_path>`
    - Explanation: This command displays the contents of the specified file on the console.
    - Steps:
        - Open your terminal.
        - Type: `hdfs dfs -cat /my_hdfs_directory/file.txt`
        - Press Enter.

Important Notes:

- **HDFS Paths:** HDFS paths always start with `/`.
- **Case Sensitivity:** HDFS paths are case-sensitive.
- **Error Handling:** If you encounter errors, check the Hadoop logs for more details.
- **Hadoop User:** Ensure you are running the commands as the Hadoop user or a user with sufficient permissions.
- **Web UI:** You can also use the Hadoop web UI (usually accessible at `http://<namenode_hostname>:9870`) to browse and manage HDFS files.

Practical Steps:

1. **Open your terminal.**
2. **Create an HDFS directory:**

    Bash

    ```
    hdfs dfs -mkdir /my_practice_dir
    ```

    - This creates a directory named `my_practice_dir` in the root of your HDFS.
3. **Upload a local file to HDFS:**

    Bash

    ```
    hdfs dfs -put ~/test.txt /my_practice_dir/uploaded_test.txt
    ```

    - This copies the `test.txt` file from your home directory to the `my_practice_dir` directory in HDFS, renaming it to `uploaded_test.txt`.

4. **List the contents of the HDFS directory:**

   Bash

   ```
   hdfs dfs -ls /my_practice_dir
   ```

   - You should see `uploaded_test.txt` listed.
5. **View the content of the uploaded file:**

   Bash

   ```
   hdfs dfs -cat /my_practice_dir/uploaded_test.txt
   ```

   - The contents of your `test.txt` file will be displayed in the terminal.
6. **Delete the uploaded file:**

   Bash

   ```
   hdfs dfs -rm /my_practice_dir/uploaded_test.txt
   ```

   - This will remove uploaded_test.txt from HDFS.
7. **Verify the file is deleted:**

   Bash

   ```
   hdfs dfs -ls /my_practice_dir
   ```

   - The file should no longer be listed.
8. **Delete the HDFS directory:**

   Bash

   ```
   hdfs dfs -rm -r /my_practice_dir
   ```

   - The `-r` option is crucial for deleting directories.
9. **Verify the directory is deleted:**

   Bash

   ```
   hdfs dfs -ls /
   ```

   - The directory `my_practice_dir` should no longer be present in the root of HDFS.

Godavari Foundation's

Godavari Institute Of Management & Research, Jalgaon

Name: __ _ _ _ _ _ Roll No: _ _ _ _ _

Date of Performance: _ _ /_ _/20_ _ Batch: _ _ _ _ _ _

Class: M.C.A. (I) Practical no: **2**

Subject: 440(B) Lab on Big Data Analytics

Title: Implement a Java program to interact with HDFS (reading and writing files).

Steps to Implement Java HDFS Interaction:

1. **Setup Hadoop:** Ensure Hadoop is running.
2. **Create Java Project:** In your IDE or with a build tool.
3. **Add Hadoop Dependency:** Include `hadoop-client` in your project's dependencies (e.g., `pom.xml` for Maven).
4. **Write Java Code:** Copy and paste the provided Java code into your project.
5. **Configure Hadoop (if needed):** Set `fs.defaultFS` in the `Configuration` object if not automatically detected.
6. **Adjust File Paths:** Modify `hdfsWritePath` and `hdfsReadPath` in the code.
7. **Compile Java Code.**
8. **Run Java Code.**
9. **Verify in HDFS:** Use `hdfs dfs -ls` and `hdfs dfs -cat` to check the file.

Code:

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.FSDataOutputStream;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class HdfsInteraction {
    public static void main(String[] args) {
        try {
            Configuration conf = new Configuration();
            // Optional: If needed, set your HDFS configuration here
            // conf.set("fs.defaultFS", "hdfs://localhost:9000");
            FileSystem fs = FileSystem.get(conf);
            // Write to HDFS
            Path writePath = new Path("/my_java_hdfs_file.txt");
            try (FSDataOutputStream out = fs.create(writePath)) {
                out.writeBytes("Hello, HDFS from Java!\n");
                out.writeBytes("Another line written via Java.\n");
                System.out.println("Written to: " + writePath);
            }
            // Read from HDFS
            Path readPath = new Path("/my_java_hdfs_file.txt");
            if (fs.exists(readPath)) {
                try (FSDataInputStream in = fs.open(readPath);
                     BufferedReader reader = new BufferedReader(new InputStreamReader(in))) {
                    String line;
                    System.out.println("Reading from: " + readPath);
                    while ((line = reader.readLine()) != null) {
                        System.out.println(line);
                    }
                }
            } else {
                System.out.println("File not found: " + readPath);
            }
            fs.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Godavari Foundation's

Godavari Institute Of Management & Research, Jalgaon

Name: __ _ _ _ _ _ _ Roll No: _ _ _ _ _

Date of Performance: _ _ /_ _/20_ _ Batch: _ _ _ _ _ _

Class: M.C.A. (I) Practical no: **3**

Subject: 440(B) Lab on Big Data Analytics

Title: Use Hadoop's built-in commands to manage files and directories.

Hadoop File/Directory Management Steps (using `hadoop fs`):

1. **Open Terminal:** Access your command-line interface.
2. **Listing Files/Directories:**
   - Type: `hadoop fs -ls <path>` (replace `<path>`)
   - Press Enter.
3. **Creating Directories:**
   - Type: `hadoop fs -mkdir -p <path>` (replace `<path>`)
   - Press Enter.
4. **Uploading Files:**
   - Type: `hadoop fs -put <local_path> <hdfs_path>` (replace paths)
   - Press Enter.
5. **Downloading Files:**
   - Type: `hadoop fs -get <hdfs_path> <local_path>` (replace paths)
   - Press Enter.
6. **Moving Files:**
   - Type: `hadoop fs -mv <source_hdfs_path> <destination_hdfs_path>` (replace paths)
   - Press Enter.
7. **Copying Files:**
   - Type: `hadoop fs -cp <source_hdfs_path> <destination_hdfs_path>` (replace paths)
   - Press Enter.
8. **Removing Files/Directories:**
   - Type: `hadoop fs -rm -r <hdfs_path>` (replace `<path>`)
   - Press Enter. (Use with caution!)
9. **Displaying File Content:**
   - Type: `hadoop fs -cat <hdfs_path>` (replace `<path>`)
   - Press Enter.

```
# Listing files/directories
hadoop fs -ls /user/yourusername/

# Creating directories
hadoop fs -mkdir -p /user/yourusername/mydir/subdir
```

```
# Uploading files
hadoop fs -put localfile.txt /user/yourusername/mydir/

# Downloading files
hadoop fs -get /user/yourusername/mydir/localfile.txt downloadedfile.txt

# Moving files
hadoop fs -mv /user/yourusername/mydir/localfile.txt /user/yourusername/newdir/

# Copying files
hadoop fs -cp /user/yourusername/mydir/localfile.txt /user/yourusername/copieddir/

# Removing files/directories
hadoop fs -rm -r /user/yourusername/mydir/

# Displaying file content
hadoop fs -cat /user/yourusername/mydir/localfile.txt
```

Godavari Foundation's

Godavari Institute Of Management & Research, Jalgaon

Name: __ _ _ _ _ _ _ Roll No: _ _ _ _ _

Date of Performance: _ _ /_ _/20_ _ Batch: _ _ _ _ _ _

Class: M.C.A. (I) Practical no: **4**

Subject: 440(B) Lab on Big Data Analytics

Title: Implement Map Side Join and Reduce Side Join.

Assumptions:

- You have a Hadoop cluster set up.
- You have two datasets that you want to join.
- You are familiar with basic MapReduce concepts.

Dataset Example:

Let's say we have two datasets:

- **users.txt:** Contains user IDs and names.
- **transactions.txt:** Contains transaction IDs, user IDs, and amounts.

1. Map-Side Join:

When to Use:

- When one of the datasets is small enough to fit into the memory of each mapper.
- When the datasets are pre-partitioned and sorted by the join key.

Steps:

1. **Prepare the Small Dataset:**
   - Ensure the smaller dataset (`users.txt` in our example) is available in HDFS.
   - If needed, pre-process the small dataset so it can be easily loaded into a data structure (e.g., a HashMap).
2. **Implement the Mapper:**
   - In the `setup()` method of your mapper, load the small dataset into a data structure (e.g., a HashMap).
   - In the `map()` method:
     - Read a record from the larger dataset (`transactions.txt`).
     - Extract the join key (user ID).
     - Look up the corresponding record in the loaded small dataset.
     - Join the records and emit the result.
3. **Configure the Job:**
   - Use the Distributed Cache to distribute the small dataset to all mappers.

   - Ensure that the input data for the larger dataset is partitioned and sorted by the join key, in the same way the small dataset is sorted.
   - Set the number of reducers to 0, because this is a map only job.
4. **Run the Job:**
   - Execute the Hadoop job.

2. Reduce-Side Join:

When to Use:

- When both datasets are large.
- When the datasets are not pre-partitioned or sorted.

Steps:

1. **Implement the Mappers:**
   - Create two mappers, one for each dataset.
   - In each mapper:
     - Read a record.
     - Extract the join key.
     - Emit the join key and the record, along with a tag indicating the dataset it came from.
2. **Implement the Reducer:**
   - The reducer receives all records with the same join key.
   - Separate the records based on their tags.
   - Perform the join operation.
   - Emit the joined records.
3. **Configure the Job:**
   - Set the input paths for both datasets.
   - Set the mapper classes.
   - Set the reducer class.
   - Set the partitioner and grouping comparator so that all keys are sent to the same reducer, and that values are grouped correctly.
4. **Run the Job:**
   - Execute the Hadoop job.

Code:

```
// Map-Side Join Code (Conceptual)

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.net.URI;
import java.util.HashMap;
import java.util.Map;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.filecache.DistributedCache;
import org.apache.hadoop.fs.Path;
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.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class MapSideJoin {

    public static class MapSideJoinMapper extends Mapper<LongWritable, Text, Text, Text> {

        private Map<String, String> users = new HashMap<>();

        @Override
        protected void setup(Context context) throws IOException, InterruptedException {
            Path[] cacheFiles = DistributedCache.getLocalCacheFiles(context.getConfiguration());
            if (cacheFiles != null && cacheFiles.length > 0) {
                String line;
                BufferedReader reader = new BufferedReader(new
    FileReader(cacheFiles[0].toString()));
                while ((line = reader.readLine()) != null) {
                    String[] parts = line.split(",");
                    users.put(parts[0], parts[1]); // userID, userName
                }
                reader.close();
            }
        }

        @Override
        protected void map(LongWritable key, Text value, Context context) throws IOException,
    InterruptedException {
            String[] parts = value.toString().split(","); // transactionID, userID, amount
            String userID = parts[1];
            String userName = users.get(userID);
            if (userName != null) {
                context.write(new Text(userID), new Text(parts[0] + "," + userName + "," +
    parts[2]));
            }
        }
    }

    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "MapSideJoin");
        job.setJarByClass(MapSideJoin.class);
        job.setMapperClass(MapSideJoinMapper.class);
        job.setNumReduceTasks(0); // Map-only job
        FileInputFormat.addInputPath(job, new Path(args[0])); // transactions.txt
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        DistributedCache.addCacheFile(new URI(args[2]), job.getConfiguration()); // users.txt
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
// Reduce-Side Join Code (Conceptual)
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
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 ReduceSideJoin {

    public static class UserMapper extends Mapper<LongWritable, Text, Text, Text> {
        @Override
        protected void map(LongWritable key, Text value, Context context) throws IOException,
     InterruptedException {
            String[] parts = value.toString().split(",");
            context.write(new Text(parts[0]), new Text("U," + parts[1])); // userID, "U,userName"
        }
    }

    public static class TransactionMapper extends Mapper<LongWritable, Text, Text, Text> {
        @Override
        protected void map(LongWritable key, Text value, Context context) throws IOException,
     InterruptedException {
            String[] parts = value.toString().split(",");
            context.write(new Text(parts[1]), new Text("T," + parts[0] + "," + parts[2])); // userID,
     "T,transactionID,amount"
        }
    }

    public static class ReduceSideJoinReducer extends Reducer<Text, Text, Text, Text> {
        @Override
        protected void reduce(Text key, Iterable<Text> values, Context context) throws
     IOException, InterruptedException {
            List<String> users = new ArrayList<>();
            List<String> transactions = new ArrayList<>();
            for (Text value : values) {
                String[] parts = value.toString().split(",");
                if (parts[0].equals("U")) {
                    users.add(parts[1]);
                } else {
                    transactions.add(parts[1] + "," + parts[2]);
                }
            }
            for (String user : users) {
                for (String transaction : transactions) {
                    context.write(key, new Text(user + "," + transaction));
                }
            }
        }
    }
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "ReduceSideJoin");
        job.setJarByClass(ReduceSideJoin.class);
        job.setMapperClass(UserMapper.class);
        job.setMapperClass(TransactionMapper.class);
        job.setReducerClass(ReduceSideJoinReducer.class);
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(Text.class);

        FileInputFormat.addInputPath(job, new Path(args[0])); // users.txt
        FileInputFormat.addInputPath(job, new Path(args[1])); // transactions.txt
        FileOutputFormat.setOutputPath(job, new Path(args[2]));

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

Godavari Foundation's

Godavari Institute Of Management & Research, Jalgaon

Name: _ Roll No: _ _ _ _ _

Date of Performance: _ _ /_ _/20_ _ Batch: _ _ _ _ _ _

Class: M.C.A. (I) Practical no: **5**

Subject: 440(B) Lab on Big Data Analytics

Title: Implement Secondary Sorting.

- **Define Composite Key:**
  - Create a Java class (e.g., `CompositeKey.java`) that implements `WritableComparable`.
  - Add fields for your natural key and secondary key.
  - Implement `compareTo()`: Compare natural keys first, then secondary keys if natural keys are equal.
  - Implement `readFields()` and `write()` for serialization.
- **Implement Mapper:**
  - Create your mapper class.
  - In the `map()` method:
    - Extract the natural key and secondary key from your input data.
    - Create an instance of your `CompositeKey` class.
    - Emit the `CompositeKey` as the key and your value.
- **Create Partitioner (Recommended):**
  - Create a custom partitioner class (e.g., `NaturalKeyPartitioner.java`) that extends `Partitioner`.
  - In the `getPartition()` method:
    - Extract only the natural key from the `CompositeKey`.
    - Use the natural key to determine the partition.
- **Create Grouping Comparator:**
  - Create a custom grouping comparator class (e.g., `NaturalKeyGroupingComparator.java`) that extends `WritableComparator`.
  - In the `compare()` method:
    - Extract only the natural key from the `CompositeKey` instances.
    - Compare only the natural keys.
- **Implement Reducer:**
  - Create your reducer class.

- In the `reduce()` method:
    - The values associated with each natural key will be sorted by the secondary key.
    - Process the sorted values.

- **Configure Hadoop Job:**

- Create a `Job` configuration.
- Set the mapper class, reducer class, and input/output formats.
- Set the `MapOutputKeyClass` to your `CompositeKey` class.
- Set the partitioner class: `job.setPartitionerClass(NaturalKeyPartitioner.class);`
- Set the grouping comparator class: `job.setGroupingComparatorClass(NaturalKeyGroupingComparator.class);`
- Set the jar by class, and input and output paths.

- **Compile and Package:**

- Compile your Java classes using `javac`.
- Create a JAR file containing your compiled classes.

- **Run the Job:**

- Use the `hadoop jar` command to run your MapReduce job on your Hadoop cluster.
- Provide the input and output paths as arguments.

```
import java.io.DataInput;
import java.io.DataOutput;
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.io.WritableComparable;
import org.apache.hadoop.io.WritableComparator;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Partitioner;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class SecondarySort {

    // Composite Key Class
    public static class CompositeKey implements WritableComparable<CompositeKey> {
        private Text naturalKey;
        private IntWritable secondaryKey;
        public CompositeKey() {
            this.naturalKey = new Text();
            this.secondaryKey = new IntWritable();
        }

        public CompositeKey(Text naturalKey, IntWritable secondaryKey) {
            this.naturalKey = naturalKey;
            this.secondaryKey = secondaryKey;
        }

        @Override
        public void write(DataOutput out) throws IOException {
            naturalKey.write(out);
            secondaryKey.write(out);
        }

        @Override
        public void readFields(DataInput in) throws IOException {
            naturalKey.readFields(in);
            secondaryKey.readFields(in);
        }

        @Override
        public int compareTo(CompositeKey other) {
            int naturalKeyComparison = this.naturalKey.compareTo(other.naturalKey);
            if (naturalKeyComparison == 0) {
                return this.secondaryKey.compareTo(other.secondaryKey);
            }
            return naturalKeyComparison;
        }

        public Text getNaturalKey() {
            return naturalKey;
        }

        public IntWritable getSecondaryKey() {
            return secondaryKey;
        }
    }

    // Mapper Class
    public static class SecondarySortMapper extends Mapper<LongWritable, Text,
CompositeKey, IntWritable> {
        private CompositeKey compositeKey = new CompositeKey();
        private IntWritable valueWritable = new IntWritable();

        @Override
        protected void map(LongWritable key, Text value, Context context) throws
IOException, InterruptedException {
            String[] parts = value.toString().split(",");
            Text naturalKey = new Text(parts[0]);
            int secondaryKey = Integer.parseInt(parts[1]);
            int valueInt = Integer.parseInt(parts[2]);

            compositeKey.naturalKey.set(naturalKey);
            compositeKey.secondaryKey.set(secondaryKey);
            valueWritable.set(valueInt);

            context.write(compositeKey, valueWritable);
        }
    }

    // Partitioner Class
    public static class NaturalKeyPartitioner extends Partitioner<CompositeKey, IntWritable>
{
        @Override
        public int getPartition(CompositeKey key, IntWritable value, int numPartitions) {
            return (key.getNaturalKey().hashCode() & Integer.MAX_VALUE) % numPartitions;
        }
    }

    // Grouping Comparator Class
    public static class NaturalKeyGroupingComparator extends WritableComparator {
        protected NaturalKeyGroupingComparator() {
            super(CompositeKey.class, true);
        }

        @Override
        public int compare(WritableComparable w1, WritableComparable w2) {
            CompositeKey key1 = (CompositeKey) w1;
            CompositeKey key2 = (CompositeKey) w2;
            return key1.getNaturalKey().compareTo(key2.getNaturalKey());
        }
    }

    // Reducer Class
    public static class SecondarySortReducer extends Reducer<CompositeKey, IntWritable,
Text, Text> {
        @Override
        protected void reduce(CompositeKey key, Iterable<IntWritable> values, Context
context) throws IOException, InterruptedException {
            StringBuilder valueString = new StringBuilder();
            for (IntWritable value : values) {
                valueString.append(value.get()).append(",");
            }
            context.write(key.getNaturalKey(), new Text(valueString.toString()));
        }
    }

    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "SecondarySort");
        job.setJarByClass(SecondarySort.class);
        job.setMapperClass(SecondarySortMapper.class);
        job.setReducerClass(SecondarySortReducer.class);
        job.setPartitionerClass(NaturalKeyPartitioner.class);
        job.setGroupingComparatorClass(NaturalKeyGroupingComparator.class);
        job.setMapOutputKeyClass(CompositeKey.class);
        job.setMapOutputValueClass(IntWritable.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);

        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

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

Godavari Foundation's

Godavari Institute Of Management & Research, Jalgaon

Name: _ Roll No: _ _ _ _ _

Date of Performance: _ _ /_ _/20_ _ Batch: _ _ _ _ _ _

Class: M.C.A. (I) Practical no: **6**

Subject: 440(B) Lab on Big Data Analytics

Title: Pipeline multiple Map Reduce jobs.

- **Develop Individual MapReduce Jobs:**
    - Create each MapReduce job as a separate Java class.
    - Ensure each job reads input from and writes output to HDFS.
    - Test each job individually to ensure it works correctly.
- **Determine Job Dependency and Order:**
    - Identify the sequence in which the jobs must execute.
    - Understand the data flow: how the output of one job serves as the input to the next.
- **Implement a Driver Class (or Script):**
    - Create a driver class (or a shell script) to manage the execution of the jobs.
    - This driver will:
        - Configure and run the first MapReduce job.
        - Wait for the first job to complete.
        - Configure and run the second MapReduce job, using the output of the first job as its input.
        - Repeat for any subsequent jobs.
- **Handle Intermediate Output:**
    - Choose a location in HDFS to store the intermediate output of each job.
    - Make sure the next job in the pipeline reads from this location.
    - Consider deleting intermediate output if it's no longer needed to save space.
- **Error Handling:**
    - Implement error checking in your driver class or script.
    - Check the return code of each job.
    - If a job fails, decide whether to stop the pipeline or take other actions.
- **Job Configuration and Parameterization:**
    - If necessary, parameterize your jobs to allow for flexible configuration.
    - Use command-line arguments or configuration files to pass parameters to the jobs.

    - This is especially helpful when dealing with different input/output paths.

- **Testing:**

    - Thoroughly test the entire pipeline.
    - Start with small datasets and gradually increase the size.
    - Verify the output of each job in the pipeline.

Code:

```
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 MapReducePipeline {

    // Job 1: Word Count
    public static class Job1Mapper extends Mapper<LongWritable, Text, Text, IntWritable> {
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();

        public void map(LongWritable key, Text value, Context context) throws java.io.IOException, InterruptedException {
            String line = value.toString();
            String[] words = line.split("\\s+");
            for (String w : words) {
                word.set(w);
                context.write(word, one);
            }
        }
    }

    public static class Job1Reducer extends Reducer<Text, IntWritable, Text, IntWritable> {
        public void reduce(Text key, Iterable<IntWritable> values, Context context) throws java.io.IOException, InterruptedException {
            int sum = 0;
            for (IntWritable val : values) {
                sum += val.get();
            }
            context.write(key, new IntWritable(sum));
        }
    }
    // Job 2: Filter words with count > 2
    public static class Job2Mapper extends Mapper<LongWritable, Text, Text,
IntWritable> {
      public void map(LongWritable key, Text value, Context context) throws
java.io.IOException, InterruptedException {
        String[] parts = value.toString().split("\\t");
        Text word = new Text(parts[0]);
        IntWritable count = new IntWritable(Integer.parseInt(parts[1]));
        context.write(word, count);
      }
    }

    public static class Job2Reducer extends Reducer<Text, IntWritable, Text, IntWritable>
{
      public void reduce(Text key, Iterable<IntWritable> values, Context context) throws
java.io.IOException, InterruptedException {
        int sum = 0;
        for (IntWritable val : values) {
          sum += val.get();
        }
        if (sum > 2) {
          context.write(key, new IntWritable(sum));
        }
      }
    }

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

      // Job 1 Configuration and Execution
      Job job1 = Job.getInstance(conf, "Word Count");
      job1.setJarByClass(MapReducePipeline.class);
      job1.setMapperClass(Job1Mapper.class);
      job1.setReducerClass(Job1Reducer.class);
      job1.setOutputKeyClass(Text.class);
      job1.setOutputValueClass(IntWritable.class);

      FileInputFormat.addInputPath(job1, new Path(args[0])); // Input text file
      FileOutputFormat.setOutputPath(job1, new Path("/intermediate/wordcount")); //
Intermediate output

      if (!job1.waitForCompletion(true)) {
        System.err.println("Job 1 failed!");
        System.exit(1);
      }

      // Job 2 Configuration and Execution
      Job job2 = Job.getInstance(conf, "Filter Words");
      job2.setJarByClass(MapReducePipeline.class);
      job2.setMapperClass(Job2Mapper.class);
      job2.setReducerClass(Job2Reducer.class);
      job2.setOutputKeyClass(Text.class);
```

```
            job2.setOutputValueClass(IntWritable.class);

            FileInputFormat.addInputPath(job2, new Path("/intermediate/wordcount")); // Input
from Job 1 output
            FileOutputFormat.setOutputPath(job2, new Path(args[1])); // Final output

            if (!job2.waitForCompletion(true)) {
                System.err.println("Job 2 failed!");
                System.exit(1);
            }

            System.exit(0);
        }
    }
```

Godavari Foundation's

Godavari Institute Of Management & Research, Jalgaon

Name: _ Roll No: _ _ _ _ _

Date of Performance: _ _ /_ _/20_ _ Batch: _ _ _ _ _ _

Class: M.C.A. (I) Practical no: **7**

Subject: 440(B) Lab on Big Data Analytics

Title: Create and use UDFs in Pig Latin scripts.

1. **Write the UDF in Java:**
   - Create a Java class that extends `org.apache.pig.EvalFunc<ReturnType>`.
   - Implement the `exec(Tuple input)` method.
   - This method takes a tuple as input and returns a value of the specified `ReturnType`.
   - Handle potential exceptions (e.g., `IOException`).
   - Compile the Java class into a JAR file.
2. **Register the JAR in Pig:**
   - In your Pig script, use the `REGISTER` command to register the JAR file containing your UDF.
   - This makes the UDF available for use in the script.
3. **Use the UDF in Your Pig Script:**
   - Call the UDF in your Pig script like any other built-in function.
   - Pass the required arguments to the UDF.

Example Scenario:

Let's create a UDF that converts a string to uppercase.

1. Create the Java UDF (UppercaseUDF.java):

Java

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

public class UppercaseUDF extends EvalFunc<String> {

    public String exec(Tuple input) throws IOException {
        if (input == null || input.size() == 0) {
            return null;
        }
        try {
            String inputString = (String) input.get(0);
            if (inputString == null) {
                return null;
            }
            return inputString.toUpperCase();
        } catch (Exception e) {
            throw new IOException("Caught exception processing input row ",
e);
        }
    }
}
```

2. Compile the Java UDF:

Bash

```
javac -classpath $(pig classpath) UppercaseUDF.java
jar cf uppercaseudf.jar UppercaseUDF.class
```

3. Use the UDF in a Pig Script (uppercase.pig):

Code snippet

```
-- Load data
data = LOAD 'input.txt' AS (text:chararray);

-- Register the UDF JAR
REGISTER 'uppercaseudf.jar';

-- Apply the UDF
uppercase_data = FOREACH data GENERATE UppercaseUDF(text);

-- Store the result
STORE uppercase_data INTO 'output';
```

Example Input Data (input.txt):

```
hello world
pig latin
hadoop ecosystem
```

4. Run the Pig Script:

Bash

```
pig uppercase.pig
```

Explanation:

- The `UppercaseUDF` class extends `EvalFunc<String>`, indicating that it returns a string.
- The `exec()` method takes a tuple as input, extracts the string, and returns its uppercase version.
- The Pig script registers the `uppercaseudf.jar` file.
- The `FOREACH` statement applies the `UppercaseUDF` to the `text` field of each record.
- The result is stored in the `output` directory.

Important Notes:

- **Error Handling:** Always include proper error handling in your UDFs to prevent unexpected failures.
- **Data Types:** Ensure that the data types in your UDF match the data types in your Pig script.

- **Complex UDFs:** For more complex UDFs, you can use other Pig interfaces like `Algebraic` and `Accumulator` to implement aggregate functions and incremental computations.
- **Testing:** Thoroughly test your UDFs before using them in production.
- **Classpath:** When compiling your UDF, make sure to include the Pig classpath to access the necessary Pig classes.
- **Data Structures:** Pig Tuples, Bags, and Maps can be accessed within the UDF.
- **Return types**: Ensure your return type matches the <Return Type> of the EvalFunc.

Code:

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

public class UppercaseUDF extends EvalFunc<String> {

    public String exec(Tuple input) throws IOException {
        if (input == null || input.size() == 0) {
            return null;
        }
        try {
            String inputString = (String) input.get(0);
            if (inputString == null) {
                return null;
            }
            return inputString.toUpperCase();
        } catch (Exception e) {
            throw new IOException("Caught exception processing input row ", e);
        }
    }
}
```

```
-- uppercase.pig
-- Load data
data = LOAD 'input.txt' AS (text:chararray);

-- Register the UDF JAR
REGISTER 'uppercaseudf.jar';

-- Apply the UDF
uppercase_data = FOREACH data GENERATE UppercaseUDF(text);

-- Store the result
STORE uppercase_data INTO 'output';
```

Godavari Foundation's

Godavari Institute Of Management & Research, Jalgaon

Name: __ _ _ _ _ _ Roll No: _ _ _ _ _

Date of Performance: _ _ /_ _/20_ _ Batch: _ _ _ _ _ _

Class: M.C.A. (I) Practical no: **8**

Subject: 440(B) Lab on Big Data Analytics

Title: Integrate UDFs to enhance the functionality of Pig scripts.

- **Identify Functionality to Enhance:**
  - Determine specific tasks in your Pig script that could benefit from custom logic.
  - Common scenarios include data validation, complex transformations, or external API calls.
- **Develop the UDF in Java:**
  - Create a Java class that extends `org.apache.pig.EvalFunc<ReturnType>`.
  - Implement the `exec(Tuple input)` method to perform the desired task.
  - Handle potential exceptions and edge cases.
  - Compile the UDF into a JAR file.
- **Register the UDF JAR in Pig:**
  - Add the `REGISTER 'path/to/your/udf.jar';` command to your Pig script.
  - Replace `'path/to/your/udf.jar'` with the actual path to your JAR file.
- **Incorporate the UDF into Your Pig Script:**
  - Call the UDF within your Pig script using the UDF's class name as a function.
  - Pass the required arguments to the UDF.
  - Use the UDF's output in subsequent Pig operations.

```
// EmailValidation.java
import java.io.IOException;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import org.apache.pig.EvalFunc;
import org.apache.pig.data.Tuple;

public class EmailValidation extends EvalFunc<String> {

    private static final Pattern EMAIL_PATTERN = Pattern.compile(
            "^[A-Za-z0-9+_.-]+@[A-Za-z0-9.-]+$");

    public String exec(Tuple input) throws IOException {
        if (input == null || input.size() == 0) {
            return null;
        }
        try {
            String email = (String) input.get(0);
            if (email == null) {
                return null;
            }
            Matcher matcher = EMAIL_PATTERN.matcher(email);
            if (matcher.matches()) {
                return email.toLowerCase();
            } else {
                return "INVALID";
            }
        } catch (Exception e) {
            throw new IOException("Error validating email", e);
        }
    }
}
```

```
-- validate_emails.pig

-- Load data

emails = LOAD 'emails.txt' AS (email:chararray);

-- Register the UDF JAR

REGISTER 'emailvalidation.jar';

-- Validate and transform emails

validated_emails = FOREACH emails GENERATE EmailValidation(email);

-- Filter out invalid emails

valid_emails = FILTER validated_emails BY $0 != 'INVALID';

-- Store the results

STORE valid_emails INTO 'valid_email_output';
```

Godavari Foundation's

Godavari Institute Of Management & Research, Jalgaon

Name: _ Roll No: _ _ _ _ _

Date of Performance: _ _ /_ _/20_ _ Batch: _ _ _ _ _ _

Class: M.C.A. (I) Practical no: **9**

Subject: 440(B) Lab on Big Data Analytics

Title: Implement and execute HiveQL queries to perform data retrieval and manipulation.

Let's implement and execute HiveQL queries to perform data retrieval and manipulation.

Steps:

1. **Start the Hive CLI:**
   - Open your terminal.
   - Type hive and press Enter. This will launch the Hive command-line interface.
2. **Create a Database (Optional):**
   - If you want to organize your tables, create a database:
     SQL
     ```
     CREATE DATABASE mydatabase;
     USE mydatabase;
     ```
3. **Create Tables:**
   - Define the structure of your data by creating tables.
   - Specify the data types and delimiters.
   - Example:
     SQL
     ```
     CREATE TABLE users (
         id INT,
         name STRING,
         age INT,
         city STRING
     )
     ROW FORMAT DELIMITED
     FIELDS TERMINATED BY ','
     STORED AS TEXTFILE;

     CREATE TABLE sales (
         sale_id INT,
         user_id INT,
         product STRING,
         amount DOUBLE
     )
     ROW FORMAT DELIMITED
     FIELDS TERMINATED BY ','
     STORED AS TEXTFILE;
     ```

4. **Load Data into Tables:**
   - Use the LOAD DATA LOCAL INPATH command to load data from your local file system into the Hive tables.
   - Example:

     SQL
     ```
     LOAD DATA LOCAL INPATH '/path/to/users.csv' OVERWRITE INTO TABLE users;
     LOAD DATA LOCAL INPATH '/path/to/sales.csv' OVERWRITE INTO TABLE sales;
     ```

   - Replace /path/to/users.csv and /path/to/sales.csv with the actual paths to your CSV files.
5. **Perform Data Retrieval (SELECT Queries):**
   - Use SELECT statements to retrieve data from the tables.
   - Examples:
     - Retrieve all data from the users table:

       SQL

       ```
       SELECT * FROM users;
       ```

     - Retrieve specific columns:

       SQL

       ```
       SELECT name, city FROM users;
       ```

     - Apply filters using WHERE clauses:

       SQL

       ```
       SELECT * FROM users WHERE age > 30;
       ```

     - Aggregate data using GROUP BY and aggregate functions:

       SQL

       ```
       SELECT city, AVG(age) FROM users GROUP BY city;
       ```

     - Join tables:

       SQL

       ```
       SELECT u.name, s.product, s.amount
       FROM users u
       JOIN sales s ON u.id = s.user_id;
       ```

     - Order the results:

       SQL

       ```
       SELECT * from sales ORDER BY amount DESC;
       ```

     - Use LIMIT to limit the number of rows:

SQL

```
SELECT * FROM users LIMIT 10;
```

6. **Perform Data Manipulation (INSERT/UPDATE/DELETE):**
   - **INSERT:**
     - Insert data from one table into another:

       SQL

       ```
       INSERT INTO TABLE new_users SELECT * FROM users WHERE age > 25;
       ```

     - Insert single rows: (Hive 0.14.0 and later)

       SQL

       ```
       INSERT INTO users VALUES (6,'John Doe',45,'London');
       ```

   - **UPDATE/DELETE:**
     - Hive does not directly support UPDATE and DELETE in the same way as traditional relational databases.
     - You can achieve similar results by creating new tables with the desired modifications or by using partitions.
     - For example, to "delete" rows, you can create a new table with only the rows you want to keep:

       SQL

       ```
       CREATE TABLE users_filtered AS SELECT * FROM users WHERE age <= 30;
       ```

     - Then you can drop the original table and rename the new table.
7. **Exit the Hive CLI:**
   - Type `quit;` or `exit;` and press Enter.

Example CSV Data (users.csv):

Code snippet

```
1,Alice,25,New York
2,Bob,32,London
3,Charlie,40,Paris
4,David,28,Tokyo
5,Eve,35,Sydney
```

Example CSV Data (sales.csv):

Code snippet

```
101,1,Laptop,1200.00
102,2,Phone,800.00
103,1,Tablet,500.00
104,3,Desktop,1500.00
105,4,Monitor,300.00
```

```
-- Start Hive CLI: hive

-- Create a database (optional)
CREATE DATABASE mydatabase;
USE mydatabase;

-- Create tables
CREATE TABLE users (
   id INT,
   name STRING,
   age INT,
   city STRING
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE;

CREATE TABLE sales (
   sale_id INT,
   user_id INT,
   product STRING,
   amount DOUBLE
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE;

-- Load data (replace paths)
LOAD DATA LOCAL INPATH '/path/to/users.csv' OVERWRITE INTO TABLE users;
LOAD DATA LOCAL INPATH '/path/to/sales.csv' OVERWRITE INTO TABLE sales;

-- Data retrieval (SELECT queries)
SELECT * FROM users;
SELECT name, city FROM users;
SELECT * FROM users WHERE age > 30;
SELECT city, AVG(age) FROM users GROUP BY city;
SELECT u.name, s.product, s.amount FROM users u JOIN sales s ON u.id = s.user_id;
SELECT * FROM sales ORDER BY amount DESC;
SELECT * FROM users LIMIT 10;

-- Data manipulation (INSERT/UPDATE/DELETE)
INSERT INTO TABLE new_users SELECT * FROM users WHERE age > 25;

-- (Hive 0.14.0 and later)
INSERT INTO users VALUES (6,'John Doe',45,'London');

-- "DELETE" (create new table)
CREATE TABLE users_filtered AS SELECT * FROM users WHERE age <= 30;

-- Exit Hive CLI
quit; -- or exit;
```

Godavari Foundation's

Godavari Institute Of Management & Research, Jalgaon

Name: __ _ _ _ _ _ _ Roll No: _ _ _ _ _

Date of Performance: _ _ /_ _/20_ _ Batch: _ _ _ _ _ _

Class: M.C.A. (I) Practical no: **10**

Subject: 440(B) Lab on Big Data Analytics

Title: Perform operations like joins, group by, and aggregations in Hive.

Let's perform joins, group by, and aggregations in Hive.

Steps:

1. **Start Hive CLI:**
   - Open your terminal.
   - Type `hive` and press Enter.
2. **Create/Use a Database (Optional):**
   - If you want to organize your tables, create a database:

     SQL

     ```
     CREATE DATABASE mydatabase;
     USE mydatabase;
     ```
3. **Create Tables:**
   - Create tables with data you want to use.
   - Example:

     SQL

     ```
     CREATE TABLE users (
         id INT,
         name STRING,
         city STRING
     )
     ROW FORMAT DELIMITED
     FIELDS TERMINATED BY ','
     STORED AS TEXTFILE;

     CREATE TABLE orders (
         order_id INT,
         user_id INT,
         product STRING,
         quantity INT,
         price DOUBLE
     )
     ROW FORMAT DELIMITED
     FIELDS TERMINATED BY ','
     ```

```
STORED AS TEXTFILE;
```

4. **Load Data:**
   - Load data into the tables from local files.
   - Example:

     SQL

     ```
     LOAD DATA LOCAL INPATH '/path/to/users.csv' OVERWRITE INTO
     TABLE users;
     LOAD DATA LOCAL INPATH '/path/to/orders.csv' OVERWRITE INTO
     TABLE orders;
     ```

   - Replace `/path/to/users.csv` and `/path/to/orders.csv` with the actual paths.
5. **Joins:**
   - **Inner Join:**

     SQL

     ```
     SELECT u.name, o.product
     FROM users u
     JOIN orders o ON u.id = o.user_id;
     ```

   - **Left Outer Join:**

     SQL

     ```
     SELECT u.name, o.product
     FROM users u
     LEFT OUTER JOIN orders o ON u.id = o.user_id;
     ```

   - **Right Outer Join:**

     SQL

     ```
     SELECT u.name, o.product
     FROM users u
     RIGHT OUTER JOIN orders o ON u.id = o.user_id;
     ```

   - **Full Outer Join:**

     SQL

     ```
     SELECT u.name, o.product
     FROM users u
     FULL OUTER JOIN orders o ON u.id = o.user_id;
     ```

6. **Group By:**
   - Group data by a column.
   - Example:

     SQL

     ```
     SELECT city, COUNT(*)
     ```

```
FROM users
GROUP BY city;
```

7. **Aggregations:**
   - Use aggregate functions with `GROUP BY`.
   - Examples:
     - **Count:**

       SQL

       ```
       SELECT product, COUNT(*)
       FROM orders
       GROUP BY product;
       ```

     - **Sum:**

       SQL

       ```
       SELECT user_id, SUM(price * quantity)
       FROM orders
       GROUP BY user_id;
       ```

     - **Average:**

       SQL

       ```
       SELECT product, AVG(price)
       FROM orders
       GROUP BY product;
       ```

     - **Max:**

       SQL

       ```
       SELECT user_id, MAX(price * quantity)
       FROM orders
       GROUP BY user_id;
       ```

     - **Min:**

       SQL

       ```
       SELECT user_id, MIN(price * quantity)
       FROM orders
       GROUP BY user_id;
       ```

     - **Combining Aggregations:**

       SQL

       ```
       SELECT city, COUNT(*), AVG(id)
       FROM users
       GROUP BY city;
       ```

8. **Filtering with `HAVING`:**

    - Use `HAVING` to filter aggregated results.
    - Example:

      SQL

      ```
      SELECT product, COUNT(*)
      FROM orders
      GROUP BY product
      HAVING COUNT(*) > 1;
      ```

9. **Exit Hive CLI:**
    - Type `quit;` or `exit;` and press Enter.

Example CSV Data (users.csv):

Code snippet

```
1,Alice,New York
2,Bob,London
3,Charlie,Paris
4,David,New York
```

Example CSV Data (orders.csv):

Code snippet

```
101,1,Laptop,1,1200.00
102,2,Phone,2,800.00
103,1,Tablet,1,500.00
104,3,Desktop,1,1500.00
105,4,Monitor,2,300.00
106,1,Mouse,3,25.00
```

```
-- Start Hive CLI: hive

-- Create/Use a database (optional)
CREATE DATABASE mydatabase;
USE mydatabase;

-- Create tables
CREATE TABLE users (
    id INT,
    name STRING,
    city STRING
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE;

CREATE TABLE orders (
    order_id INT,
    user_id INT,
    product STRING,
```

```
    quantity INT,
    price DOUBLE
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE;

-- Load data (replace paths)
LOAD DATA LOCAL INPATH '/path/to/users.csv' OVERWRITE INTO TABLE users;
LOAD DATA LOCAL INPATH '/path/to/orders.csv' OVERWRITE INTO TABLE orders;

-- Joins
SELECT u.name, o.product FROM users u JOIN orders o ON u.id = o.user_id;
SELECT u.name, o.product FROM users u LEFT OUTER JOIN orders o ON u.id =
o.user_id;
SELECT u.name, o.product FROM users u RIGHT OUTER JOIN orders o ON u.id =
o.user_id;
SELECT u.name, o.product FROM users u FULL OUTER JOIN orders o ON u.id =
o.user_id;

-- Group By
SELECT city, COUNT(*) FROM users GROUP BY city;

-- Aggregations
SELECT product, COUNT(*) FROM orders GROUP BY product;
SELECT user_id, SUM(price * quantity) FROM orders GROUP BY user_id;
SELECT product, AVG(price) FROM orders GROUP BY product;
SELECT user_id, MAX(price * quantity) FROM orders GROUP BY user_id;
SELECT user_id, MIN(price * quantity) FROM orders GROUP BY user_id;
SELECT city, COUNT(*), AVG(id) FROM users GROUP BY city;

-- Filtering with HAVING
SELECT product, COUNT(*) FROM orders GROUP BY product HAVING COUNT(*) > 1;

-- Exit Hive CLI
quit; -- or exit;
```