1.Perform the following HDFS operations using appropriate commands:

1.Upload a large text file from your local system to HDFS.

2.List all the files in your HDFS directory.

3.Retrieve and view the content of a file stored in HDFS. Implement a simple MapReduce Word Count program without a combiner and measure its performance.

4.Modify the program to use a combiner.

5.Compare the performance before and after adding the combiner.

6.Discuss the results and explain why the combiner improved performance.

Sol1. Upload a large text file from your local system to HDFS

Command:

bash
Copy code
```
hdfs dfs -put /path/to/localfile.txt /user/hadoop/dir/
```

Explanation:

- `/path/to/localfile.txt`: The path to the local text file you want to upload.
- `/user/hadoop/dir/`: The destination directory in HDFS.

2. List all the files in your HDFS directory

Command:

bash
Copy code
```
hdfs dfs -ls /user/hadoop/dir/
```

Explanation:

- Lists all files in the specified HDFS directory.

3. Retrieve and view the content of a file stored in HDFS

Command:

bash
Copy code
```
hdfs dfs -cat /user/hadoop/dir/filename.txt
```

Explanation:

- Retrieves and prints the content of `filename.txt` from the specified HDFS directory.

---

4. Implement a simple MapReduce Word Count program without a combiner

Steps to implement:

1. **Create a Java program** (WordCount.java):

```
java
Copy code
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.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 {
            String[] words = value.toString().split("\\s+");
            for (String w : words) {
                word.set(w);
                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();
            }
```

```java
                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.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);
    }
}
```

Example commands:

1. **Compile the program**:

   bash
   Copy code
   ```bash
   javac -cp $(hadoop classpath) -d . WordCount.java
   ```

2. **Create a jar file**:

   bash
   Copy code
   ```bash
   jar -cvf wordcount.jar -C . .
   ```

3. **Run the MapReduce job**:

   bash
   Copy code
   ```bash
   hadoop jar wordcount.jar WordCount /user/hadoop/input /user/hadoop/output
   ```

   - `/user/hadoop/input`: Input directory in HDFS.
   - `/user/hadoop/output`: Output directory in HDFS.

Performance measurement:

You can measure the performance by analyzing the job completion time, visible in the job tracker (or ResourceManager UI).

---

5. Modify the program to use a combiner

Modify the WordCount program to use a combiner:

In the main class, add the following line to set the combiner:

```
java
Copy code
job.setCombinerClass(IntSumReducer.class);
```

Example command (same as before, but with the combiner now enabled):

```
bash
Copy code
hadoop jar wordcount.jar WordCount /user/hadoop/input
/user/hadoop/output_with_combiner
```

6. Compare the performance before and after adding the combiner

- **Without combiner**: The mapper outputs every occurrence of each word, and all these outputs are transferred to the reducer. This increases the amount of data shuffled across the network.
- **With combiner**: The combiner acts as a mini-reducer at the map stage, reducing the amount of data transferred to the reducer by aggregating counts locally before sending them over the network.

7. Discussion on why the combiner improved performance

- **Without combiner**: Every single occurrence of a word is sent from the mapper to the reducer, which results in a higher amount of intermediate data being transferred, leading to increased network and disk I/O.
- **With combiner**: The combiner reduces the intermediate data by aggregating locally on the mapper nodes. This significantly reduces the amount of data shuffling between the mapper and reducer stages, thereby improving performance, especially when the input dataset is large.

2.Implement a MapReduce program using TextInputFormat to read a large text file and count the frequency of key-value pairs where each line contains a key and a value.

2solProblem Outline:

- Each line in the input text file consists of a key and a value (e.g., `key value`).

- The goal is to count the occurrences of each key-value pair.

MapReduce Program

1. Mapper Class:

The `Mapper` will split each line into key and value pairs and emit them as output.

2. Reducer Class:

The `Reducer` will aggregate the key-value pairs and count their occurrences.

3. Driver Class:

This class will set up the job configuration and specify input/output formats.

Code Implementation

Step 1: Create the Mapper Class

```
java
Copy code
import java.io.IOException;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class KeyValueMapper extends Mapper<LongWritable, Text, Text, Text> {
    private Text keyText = new Text();
    private Text valueText = new Text();

    @Override
    protected void map(LongWritable key, Text value, Context context) throws
IOException, InterruptedException {
        // Each line is expected to be in the format: key value
        String[] line = value.toString().split("\\s+");
        if (line.length == 2) { // Ensure the line contains exactly two words
            keyText.set(line[0]);
            valueText.set(line[1]);
            context.write(keyText, valueText);
        }
    }
}
```

Step 2: Create the Reducer Class

```
java
Copy code
import java.io.IOException;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;
```

```java
public class KeyValueReducer extends Reducer<Text, Text, Text, Text> {
    @Override
    protected void reduce(Text key, Iterable<Text> values, Context context)
throws IOException, InterruptedException {
        // Count the occurrences of each unique value for a given key
        Map<String, Integer> valueCountMap = new HashMap<>();
        for (Text value : values) {
            String val = value.toString();
            valueCountMap.put(val, valueCountMap.getOrDefault(val, 0) + 1);
        }

        // Write each key-value pair and their counts
        for (Map.Entry<String, Integer> entry : valueCountMap.entrySet()) {
            context.write(new Text(key.toString() + " " + entry.getKey()), new
Text(entry.getValue().toString()));
        }
    }
}
```

Step 3: Create the Driver Class

```java
java
Copy code
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class KeyValueCount {
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "Key Value Count");

        job.setJarByClass(KeyValueCount.class);
        job.setMapperClass(KeyValueMapper.class);
        job.setReducerClass(KeyValueReducer.class);

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

        // Specify the TextInputFormat class to read key-value pairs from the
text file

job.setInputFormatClass(org.apache.hadoop.mapreduce.lib.input.TextInputFormat.c
lass);

        // Input and output paths
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
```

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

Steps to Run the MapReduce Program

1. Compile the Java Code

```
bash
Copy code
javac -cp $(hadoop classpath) -d . KeyValueMapper.java KeyValueReducer.java
KeyValueCount.java
```

2. Create a JAR File

```
bash
Copy code
jar -cvf keyvaluecount.jar -C . .
```

3. Upload the Input File to HDFS

Assuming the input file (`input.txt`) is on your local system, upload it to HDFS:

```
bash
Copy code
hdfs dfs -put /path/to/input.txt /user/hadoop/input/
```

Example `input.txt` content:

```
Copy code
apple 1
banana 2
apple 1
orange 3
banana 2
```

4. Run the MapReduce Job

```
bash
Copy code
hadoop jar keyvaluecount.jar KeyValueCount /user/hadoop/input
/user/hadoop/output
```

- `/user/hadoop/input`: The directory containing the input file in HDFS.
- `/user/hadoop/output`: The directory where the output will be written.

5. View the Output

Once the job is completed, check the output in HDFS:

```bash
hdfs dfs -cat /user/hadoop/output/part-r-00000
```

Expected Output:

```
apple 1 2
banana 2 2
orange 3 1
```

Explanation

- **Mapper**: The input file is read line by line, where each line contains a key-value pair. The mapper outputs the key and value as a pair.
- **Reducer**: For each key, the reducer counts the frequency of each unique value.
- **TextInputFormat**: Used to read lines from the input file, splitting them into key-value pairs.

This program will count the occurrences of each unique key-value pair from the text file.

3.Implement a MapReduce program using NLineInputFormat. Configure the program to process N lines per map task and observe the difference in task distribution.

ChatGPT said:

3sol:Implementing a MapReduce Program Using `NLineInputFormat`

`NLineInputFormat` is a specialized input format that splits the input file based on a specified number of lines per map task. This can be useful when you want to control the granularity of your MapReduce tasks and improve performance by adjusting the load for each mapper.

Steps to Implement and Configure the Program

We will create a MapReduce program similar to the Word Count example but configure it to use `NLineInputFormat` and process **N lines per map task**.

Code Implementation

Step 1: Create the Mapper Class

In this example, the mapper processes `N` lines at a time, splits them into words, and counts the occurrences of each word.

java

```
Copy code
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class NLineMapper extends Mapper<LongWritable, Text, Text, IntWritable>
{
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    @Override
    protected void map(LongWritable key, Text value, Context context) throws
IOException, InterruptedException {
        String[] lines = value.toString().split("\\s+");
        for (String wordText : lines) {
            word.set(wordText);
            context.write(word, one);
        }
    }
}
```

Step 2: Create the Reducer Class

The reducer aggregates the word counts emitted by each mapper.

```
java
Copy code
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class NLineReducer 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));
    }
}
```

Step 3: Create the Driver Class

The driver class sets up the job configuration to use `NLineInputFormat` and specify the number of lines per map task.

java
Copy code
```java
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.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.NLineInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class NLineWordCount {
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "N Line Word Count");

        // Set classes
        job.setJarByClass(NLineWordCount.class);
        job.setMapperClass(NLineMapper.class);
        job.setReducerClass(NLineReducer.class);

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

        // Use NLineInputFormat
        job.setInputFormatClass(NLineInputFormat.class);

        // Set number of lines per split (e.g., 5 lines per map task)
        NLineInputFormat.setNumLinesPerSplit(job, 5);

        // Specify input and output paths
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

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

Steps to Run the MapReduce Program

1. Compile the Java Code

bash
Copy code
```bash
javac -cp $(hadoop classpath) -d . NLineMapper.java NLineReducer.java
NLineWordCount.java
```

2. Create a JAR File

bash
Copy code
```bash
jar -cvf nlinewordcount.jar -C . .
```

3. Upload the Input File to HDFS

If the input file (`input.txt`) is on your local system, upload it to HDFS:

```
bash
Copy code
hdfs dfs -put /path/to/input.txt /user/hadoop/input/
```

4. Run the MapReduce Job with `NLineInputFormat`

```
bash
Copy code
hadoop jar nlinewordcount.jar NLineWordCount /user/hadoop/input
/user/hadoop/output_nline
```

- `/user/hadoop/input`: Directory containing the input file in HDFS.
- `/user/hadoop/output_nline`: Output directory where results will be written.

5. View the Output

Once the job is completed, check the output in HDFS:

```
bash
Copy code
hdfs dfs -cat /user/hadoop/output_nline/part-r-00000
```

Example Input File (input.txt)

```
Copy code
apple banana apple
orange banana apple
banana apple orange
grape apple banana
orange grape apple
```

Example Output (part-r-00000)

```
Copy code
apple 8
banana 5
grape 2
orange 3
```

Key Points on Task Distribution

- By using `NLineInputFormat`, the input is split into chunks where each mapper processes `N` lines (in this case, 5 lines per task).

- If the input file contains 100 lines and you set `NLineInputFormat` to process 10 lines per map task, the system will launch 10 map tasks.
- The more lines you assign to each map task, the fewer map tasks will be created, which may reduce overhead but can also increase the load on each mapper.

Comparing Task Distribution:

- **Without `NLineInputFormat`**: By default, the number of mappers is determined by the size of the input split, usually based on block size (e.g., 64MB or 128MB).
- **With `NLineInputFormat`**: You can control how many lines each map task processes, allowing for more granular task distribution. This can improve performance if processing smaller splits is more efficient for your data.

Observation:

1. **Fine-grained control**: Using `NLineInputFormat`, you can split the data based on the number of lines, which helps when dealing with line-based records in text files.
2. **Efficiency**: The task distribution is more balanced when controlling the number of lines per task, reducing uneven workloads across map tasks.
3. **Performance**: Depending on the dataset size and processing complexity, adjusting the number of lines per split can improve the overall execution time by optimizing the load across the mappers.

4.Modify the MapReduce program to use MultilineInputFormat, where a record spans multiple lines (e.g., JSON or XML records), and count the occurrences of a specific tag or keyword.

4solModifying the MapReduce Program to Use `MultilineInputFormat`

`MultilineInputFormat` is useful when a single record spans multiple lines, such as when processing files in formats like JSON, XML, or any other structured format where records may not be confined to one line. The goal of this program is to read multi-line records and count occurrences of a specific tag or keyword.

Problem Outline:

- The input contains multi-line records (e.g., XML or JSON).
- We want to count the occurrences of a specific tag or keyword across all the records.

Approach:

- Use a custom `MultilineInputFormat` to handle multi-line records.
- Implement a `Mapper` class to parse these records and count the occurrences of a specific keyword or tag.
- Use a `Reducer` class to aggregate the keyword occurrences across the mappers.

Example Scenario:

Let's assume the input is in an XML format, where each record spans multiple lines, and we want to count the occurrences of the `<item>` tag.

Code Implementation

Step 1: Create a Custom `MultilineInputFormat`

`MultilineInputFormat` can be implemented by extending the default `TextInputFormat`. We'll override the logic to group multiple lines into one record. For example, in the case of XML, each record can be bounded by a start tag (`<item>`) and an end tag (`</item>`).

java
Copy code
```
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.InputSplit;
import org.apache.hadoop.mapreduce.RecordReader;
import org.apache.hadoop.mapreduce.TaskAttemptContext;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;

public class XMLInputFormat extends TextInputFormat {
    @Override
    public RecordReader<LongWritable, Text> createRecordReader(InputSplit
split, TaskAttemptContext context) {
        return new XMLRecordReader();
    }
}
```

Step 2: Create the `XMLRecordReader` Class

This class reads multi-line XML records by identifying start and end tags. Modify the tags according to your use case (e.g., JSON or custom format).

java
Copy code
```
import java.io.IOException;
import org.apache.hadoop.fs.FSDataInputStream;
```

```java
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.InputSplit;
import org.apache.hadoop.mapreduce.RecordReader;
import org.apache.hadoop.mapreduce.TaskAttemptContext;
import org.apache.hadoop.mapreduce.lib.input.FileSplit;
import org.apache.hadoop.util.LineReader;

public class XMLRecordReader extends RecordReader<LongWritable, Text> {
    private FSDataInputStream inputStream;
    private LineReader lineReader;
    private LongWritable currentKey = new LongWritable();
    private Text currentValue = new Text();
    private String startTag = "<item>";
    private String endTag = "</item>";

    @Override
    public void initialize(InputSplit inputSplit, TaskAttemptContext context)
throws IOException {
        Path file = ((FileSplit) inputSplit).getPath();
        inputStream =
file.getFileSystem(context.getConfiguration()).open(file);
        lineReader = new LineReader(inputStream);
    }

    @Override
    public boolean nextKeyValue() throws IOException {
        Text line = new Text();
        StringBuilder xmlRecord = new StringBuilder();
        boolean insideRecord = false;

        while (lineReader.readLine(line) > 0) {
            String lineStr = line.toString().trim();

            if (lineStr.contains(startTag)) {
                insideRecord = true;
            }

            if (insideRecord) {
                xmlRecord.append(lineStr).append("\n");
            }

            if (lineStr.contains(endTag)) {
                insideRecord = false;
                currentKey.set(inputStream.getPos());
                currentValue.set(xmlRecord.toString());
                return true;
            }
        }
        return false;
```

```
    }

    @Override
    public LongWritable getCurrentKey() {
        return currentKey;
    }

    @Override
    public Text getCurrentValue() {
        return currentValue;
    }

    @Override
    public float getProgress() throws IOException {
        return inputStream.getPos();
    }

    @Override
    public void close() throws IOException {
        inputStream.close();
    }
}
```

Step 3: Create the Mapper Class

The mapper reads each multi-line record and looks for the specific tag or keyword (e.g., `<item>`).

```java
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class XMLMapper extends Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text keyword = new Text();

    @Override
    protected void setup(Context context) throws IOException,
InterruptedException {
        // The keyword/tag to count (e.g., <item>)
        keyword.set(context.getConfiguration().get("xml.tag", "<item>"));
    }

    @Override
    protected void map(LongWritable key, Text value, Context context) throws
IOException, InterruptedException {
        String xmlRecord = value.toString();
```

```java
        // Count occurrences of the keyword/tag in the record
        int count = xmlRecord.split(keyword.toString(), -1).length - 1;

        if (count > 0) {
            context.write(new Text(keyword), new IntWritable(count));
        }
    }
}
```

Step 4: Create the Reducer Class

The reducer aggregates the counts of the keyword occurrences emitted by the mappers.

java
Copy code
```java
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class XMLReducer 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));
    }
}
```

Step 5: Create the Driver Class

The driver class configures the job and sets the input format to the custom `XMLInputFormat`.

java
Copy code
```java
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.lib.output.FileOutputFormat;

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

        // Set the tag or keyword to count
        conf.set("xml.tag", "<item>");
```

```java

        Job job = Job.getInstance(conf, "XML Tag Count");

        job.setJarByClass(XMLWordCount.class);
        job.setMapperClass(XMLMapper.class);
        job.setReducerClass(XMLReducer.class);

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

        // Set the custom XMLInputFormat
        job.setInputFormatClass(XMLInputFormat.class);

        // Set input and output paths
        XMLInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

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

Steps to Run the MapReduce Program

1. Compile the Java Code

bash
Copy code
```bash
javac -cp $(hadoop classpath) -d . XMLInputFormat.java XMLRecordReader.java
XMLMapper.java XMLReducer.java XMLWordCount.java
```

2. Create a JAR File

bash
Copy code
```bash
jar -cvf xmltagcount.jar -C . .
```

3. Upload the Input File to HDFS

If the input file (e.g., `input.xml`) is on your local system, upload it to HDFS:

bash
Copy code
```bash
hdfs dfs -put /path/to/input.xml /user/hadoop/input/
```

Example XML Input File (`input.xml`):

xml
Copy code
```xml
<items>
  <item>
    <name>Apple</name>
```

```
    <price>50</price>
  </item>
  <item>
    <name>Banana</name>
    <price>20</price>
  </item>
  <item>
    <name>Orange</name>
    <price>30</price>
  </item>
</items>
```

4. Run the MapReduce Job

bash
Copy code
```bash
hadoop jar xmltagcount.jar XMLWordCount /user/hadoop/input
/user/hadoop/output_xml
```

- `/user/hadoop/input`: Directory containing the input XML file in HDFS.
- `/user/hadoop/output_xml`: Output directory where results will be written.

5. View the Output

Once the job is completed, check the output in HDFS:

bash
Copy code
```bash
hdfs dfs -cat /user/hadoop/output_xml/part-r-00000
```

Expected Output (if counting occurrences of `<item>`):

php
Copy code
```
<item> 3
```

5.Convert a text file to a SequenceFile and implement a MapReduce program that reads data from the sequence file

5sol:Converting a Text File to a SequenceFile and Implementing a MapReduce Program to Read from It

In Hadoop, a **SequenceFile** is a binary file format optimized for key-value pairs. It is often used for intermediate outputs between MapReduce jobs. Here, we'll go through two main parts:

1. **Converting a text file to a SequenceFile**.
2. **Writing and running a MapReduce program that reads from the SequenceFile**.

Part 1: Converting a Text File to a SequenceFile

Assume you have a text file where each line consists of a key-value pair separated by a delimiter. We'll convert this file into a Hadoop **SequenceFile**.

Step 1: Create a Sample Input Text File

Let's create a sample text file (`input.txt`) containing key-value pairs:

```
mathematica
Copy code
1,Apple
2,Banana
3,Orange
4,Mango
```

Step 2: Hadoop Code to Convert Text to a SequenceFile

Use the following code to read the text file and convert it into a SequenceFile, where the key will be the line number (or other numeric identifier) and the value will be the rest of the line.

Java Code to Convert Text to SequenceFile:

```
java
Copy code
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.Text;
import org.apache.hadoop.io.SequenceFile;
import org.apache.hadoop.io.SequenceFile.Writer;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.util.ReflectionUtils;

public class TextToSequenceFile {
    public static void main(String[] args) throws IOException {
        Configuration conf = new Configuration();
        FileSystem fs = FileSystem.get(conf);

        Path inputFile = new Path(args[0]);  // Input text file path
        Path outputFile = new Path(args[1]); // Output SequenceFile path

        Writer writer = null;
        try {
            writer = SequenceFile.createWriter(conf,
                    Writer.file(outputFile),
                    Writer.keyClass(IntWritable.class),
                    Writer.valueClass(Text.class));

            // Read the input text file line by line
```

```
            FSDataInputStream inStream = fs.open(inputFile);
            int lineNum = 1;
            String line;

            while ((line = inStream.readLine()) != null) {
                writer.append(new IntWritable(lineNum), new Text(line));
                lineNum++;
            }
        } finally {
            if (writer != null) {
                writer.close();
            }
        }
    }
}
```

Step 3: Compile and Run the Program

1. Compile the code:

```
bash
Copy code
javac -cp $(hadoop classpath) TextToSequenceFile.java
jar -cvf texttoseq.jar TextToSequenceFile*.class
```

2. Run the program:

```
bash
Copy code
hadoop jar texttoseq.jar TextToSequenceFile /path/to/input.txt
/path/to/output.seq
```

This will convert the text file into a Hadoop **SequenceFile** and store it as `output.seq` in HDFS.

Verify the SequenceFile

You can check the content of the SequenceFile using:

```
bash
Copy code
hadoop fs -text /path/to/output.seq
```

Part 2: Writing a MapReduce Program to Read from a SequenceFile

Now that we have the **SequenceFile**, let's write a MapReduce program that reads data from this file and processes it.

Step 1: Mapper to Read from SequenceFile

The **Mapper** will process the key-value pairs from the **SequenceFile**. The key is the line number, and the value is the text from each line of the original file.

```
java
Copy code
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class SequenceFileMapper extends Mapper<IntWritable, Text, Text, 
IntWritable> {
    private final static IntWritable one = new IntWritable(1);

    @Override
    public void map(IntWritable key, Text value, Context context) throws 
IOException, InterruptedException {
        // Split the value into key-value parts
        String[] parts = value.toString().split(",");
        if (parts.length == 2) {
            context.write(new Text(parts[1]), one);  // Emit the fruit name and 
count 1
        }
    }
}
```

Step 2: Reducer to Aggregate the Results

The **Reducer** will count how many times each fruit appears in the file.

```
java
Copy code
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class SequenceFileReducer extends Reducer<Text, IntWritable, Text, 
IntWritable> {
    @Override
    public 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));  // Emit fruit and its count
    }
}
```

Step 3: Driver Code to Set Up the Job

```java
java
Copy code
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.lib.input.SequenceFileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class SequenceFileJob {
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "SequenceFile Word Count");

        job.setJarByClass(SequenceFileJob.class);
        job.setMapperClass(SequenceFileMapper.class);
        job.setReducerClass(SequenceFileReducer.class);

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

        // Set input format to SequenceFile
        job.setInputFormatClass(SequenceFileInputFormat.class);

        // Set input and output paths
        SequenceFileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

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

Step 4: Compile and Run the Program

1. **Compile the code**:

```bash
bash
Copy code
javac -cp $(hadoop classpath) -d . SequenceFileMapper.java
SequenceFileReducer.java SequenceFileJob.java
jar -cvf sequencefilejob.jar -C . .
```

2. **Run the MapReduce job**:

```bash
bash
Copy code
hadoop jar sequencefilejob.jar SequenceFileJob /path/to/output.seq
/path/to/seqfile_output
```

Step 5: View the Output

After the job completes, view the results:

```
bash
Copy code
hadoop fs -cat /path/to/seqfile_output/part-r-00000
```

Expected Output (if using the sample input):

```
mathematica
Copy code
Apple   1
Banana  1
Mango   1
Orange  1
```

Explanation

1. **Conversion to SequenceFile**: We converted a text file into a binary SequenceFile where each record is stored as a key-value pair (the key being the line number and the value being the text).
2. **MapReduce Program**: We wrote a MapReduce program that reads from this SequenceFile using `SequenceFileInputFormat`, processes the key-value pairs, and counts the frequency of values (in this case, fruit names).

This approach is more efficient for larger datasets since SequenceFiles are binary and more compact than text files, resulting in faster I/O operations.

6Implement secondary sorting in MapReduce by writing a job that sorts the data first by key and then by value before passing it to the reducer.6sol

6solProblem Scenario

We will write a MapReduce job where data is sorted first by the **key** and then by the **value** before being passed to the reducer. For example, consider a dataset containing pairs of cities and temperatures:

```
Copy code
NewYork,15
NewYork,10
NewYork,20
Chicago,25
Chicago,22
Chicago,19
```

Here, the **key** is the city name and the **value** is the temperature. We want to sort the data first by the city (key) and then by temperature (value), so the reducer will receive temperatures in sorted order for each city.

Steps to Implement Secondary Sorting in MapReduce

1. **Custom Writable for Key**: We'll need to define a custom composite key class to use both the city and temperature.
2. **Custom Partitioner**: Ensure all data for a given city is sent to the same reducer.
3. **Custom Comparator**: Ensure data is sorted first by city and then by temperature.
4. **Mapper and Reducer**: Write the mapper and reducer to process the sorted data.

Step 1: Define a Custom Writable Comparable Class

The custom key will consist of the city (String) and temperature (Integer).

java
Copy code

```
import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;
import org.apache.hadoop.io.WritableComparable;

public class CityTemperatureKey implements
WritableComparable<CityTemperatureKey> {
    private String city;
    private int temperature;

    public CityTemperatureKey() {}

    public CityTemperatureKey(String city, int temperature) {
        this.city = city;
        this.temperature = temperature;
    }

    public String getCity() {
        return city;
    }

    public int getTemperature() {
        return temperature;
    }

    @Override
    public void write(DataOutput out) throws IOException {
        out.writeUTF(city);
        out.writeInt(temperature);
    }
```

```java
    @Override
    public void readFields(DataInput in) throws IOException {
        city = in.readUTF();
        temperature = in.readInt();
    }

    @Override
    public int compareTo(CityTemperatureKey o) {
        int result = this.city.compareTo(o.city);
        if (result == 0) {
            // If cities are the same, compare by temperature (ascending order)
            result = Integer.compare(this.temperature, o.temperature);
        }
        return result;
    }

    @Override
    public String toString() {
        return city + "," + temperature;
    }
}
```

Step 2: Define the Mapper

The mapper will output the city as the key and the temperature as the value, wrapped in the `CityTemperatureKey` class.

java
Copy code
```java
import java.io.IOException;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class SecondarySortMapper extends Mapper<LongWritable, Text, 
CityTemperatureKey, Text> {

    @Override
    protected void map(LongWritable key, Text value, Context context) throws 
IOException, InterruptedException {
        // Split input into city and temperature
        String[] fields = value.toString().split(",");
        String city = fields[0];
        int temperature = Integer.parseInt(fields[1]);

        // Emit city and temperature as composite key
        context.write(new CityTemperatureKey(city, temperature), new 
Text(city));
    }
}
```

Step 3: Define the Reducer

The reducer will receive the data already sorted by the composite key. We can simply output the city and temperature.

```java
java
Copy code
import java.io.IOException;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class SecondarySortReducer extends Reducer<CityTemperatureKey, Text,
Text, Text> {

    @Override
    protected void reduce(CityTemperatureKey key, Iterable<Text> values,
Context context)
            throws IOException, InterruptedException {
        // For each city, emit the temperature in sorted order
        for (Text value : values) {
            context.write(new Text(key.getCity()), new
Text(String.valueOf(key.getTemperature())));
        }
    }
}
```

Step 4: Define a Custom Partitioner

This will ensure that all records for the same city (key) go to the same reducer, even if they have different temperatures.

```java
java
Copy code
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Partitioner;

public class CityPartitioner extends Partitioner<CityTemperatureKey, Text> {

    @Override
    public int getPartition(CityTemperatureKey key, Text value, int
numReduceTasks) {
        return Math.abs(key.getCity().hashCode() % numReduceTasks);
    }
}
```

Step 5: Define a Custom Grouping Comparator

This ensures that the reducer receives all values for the same city together, even if the temperatures are different.

```java
import org.apache.hadoop.io.WritableComparable;
import org.apache.hadoop.io.WritableComparator;

public class CityGroupingComparator extends WritableComparator {

    protected CityGroupingComparator() {
        super(CityTemperatureKey.class, true);
    }

    @Override
    public int compare(WritableComparable w1, WritableComparable w2) {
        CityTemperatureKey k1 = (CityTemperatureKey) w1;
        CityTemperatureKey k2 = (CityTemperatureKey) w2;
        return k1.getCity().compareTo(k2.getCity());
    }
}
```

Step 6: Driver Code

Finally, the driver code will set up the job with all the custom components we've defined.

```java
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class SecondarySortDriver {

    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "Secondary Sort");

        job.setJarByClass(SecondarySortDriver.class);
        job.setMapperClass(SecondarySortMapper.class);
        job.setReducerClass(SecondarySortReducer.class);

        // Set custom key and value classes
        job.setMapOutputKeyClass(CityTemperatureKey.class);
        job.setMapOutputValueClass(Text.class);

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

        // Set partitioner, grouping comparator, and sort comparator
        job.setPartitionerClass(CityPartitioner.class);
        job.setGroupingComparatorClass(CityGroupingComparator.class);
```

```
        // Set input and output paths
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

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

Step 7: Compile and Run the Job

1. **Compile the code**:

```
bash
Copy code
javac -cp $(hadoop classpath) *.java
jar -cvf secondarysort.jar *.class
```

2. **Run the job**:

```
bash
Copy code
hadoop jar secondarysort.jar SecondarySortDriver /path/to/input /path/to/output
```

Step 8: Input and Output

Input (Text File)

```
Copy code
NewYork,15
NewYork,10
NewYork,20
Chicago,25
Chicago,22
Chicago,19
```

Output (Sorted by City and Temperature)

```
Copy code
Chicago    19
Chicago    22
Chicago    25
NewYork    10
NewYork    15
NewYork    20
```

7.

Implement a MapReduce program that uses Distributed Cache to perform a Map-Side Join. Use a small dataset that will be cached on each node and join it with a larger dataset.

7sol

7solProblem Scenario

Let's assume we have two datasets:

1. **Small dataset** (`departments.txt`) that contains department information:

```
Copy code
1,Engineering
2,HR
3,Sales
```

2. **Large dataset** (`employees.txt`) that contains employee information:

```
Copy code
101,John,1
102,Alice,2
103,Bob,1
104,Eve,3
```

We want to join these datasets on the **department ID** to create a new dataset that shows the employee names along with their department names.

Step 1: Prepare the Datasets

- **Small Dataset (departments.txt)**:

```
Copy code
1,Engineering
2,HR
3,Sales
```

- **Large Dataset (employees.txt)**:

```
Copy code
101,John,1
102,Alice,2
103,Bob,1
104,Eve,3
```

Step 2: Implement the MapReduce Program

1. Mapper Class

The mapper will:

- Load the small dataset (departments) into memory using the **Distributed Cache**.
- Read each employee record from the large dataset and join it with the department name based on the department ID.

```
java
Copy code
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.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

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

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

    @Override
    protected void setup(Context context) throws IOException {
        URI[] cacheFiles = context.getCacheFiles();
        if (cacheFiles != null && cacheFiles.length > 0) {
            for (URI cacheFile : cacheFiles) {
                loadDepartmentData(new Path(cacheFile.getPath()).getName());
            }
        }
    }

    private void loadDepartmentData(String fileName) throws IOException {
        BufferedReader reader = new BufferedReader(new FileReader(fileName));
        String line;
        while ((line = reader.readLine()) != null) {
            String[] fields = line.split(",");
            if (fields.length == 2) {
                departmentMap.put(fields[0], fields[1]);  // Key: department
ID, Value: department name
            }
        }
        reader.close();
```

```
    }

    @Override
    protected void map(LongWritable key, Text value, Context context) throws
IOException, InterruptedException {
        String[] fields = value.toString().split(",");
        if (fields.length == 3) {
            String empId = fields[0];
            String empName = fields[1];
            String deptId = fields[2];

            String deptName = departmentMap.get(deptId);
            if (deptName != null) {
                context.write(new Text(empId), new Text(empName + "," +
deptName));
            }
        }
    }
}
```

2. Driver Class

The driver class configures the job to use the distributed cache and sets up the input and output paths.

```
java
Copy code
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class MapSideJoinDriver {
    public static void main(String[] args) throws Exception {
        if (args.length != 3) {
            System.err.println("Usage: MapSideJoinDriver <input path> <output
path> <cache file path>");
            System.exit(-1);
        }

        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "Map-Side Join with Distributed
Cache");

        job.setJarByClass(MapSideJoinDriver.class);
        job.setMapperClass(MapSideJoinMapper.class);
        job.setNumReduceTasks(0);  // No need for a reducer in map-side join

        job.setOutputKeyClass(Text.class);
```

```
        job.setOutputValueClass(Text.class);

        // Add the small dataset (departments) to the distributed cache
        job.addCacheFile(new Path(args[2]).toUri());

        // Set input and output paths
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

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

Step 3: Compile and Package the Program

1. **Compile the Java files**:

```
bash
Copy code
javac -cp $(hadoop classpath) -d . *.java
```

2. **Create a JAR file**:

```
bash
Copy code
jar -cvf mapsidejoin.jar -C . .
```

Step 4: Prepare the Input Files

Assume the input files (`employees.txt` and `departments.txt`) are in your local file system.

Upload the `employees.txt` (large dataset) to HDFS:

```
bash
Copy code
hdfs dfs -put employees.txt /input/employees.txt
```

Upload the `departments.txt` (small dataset) to HDFS for use in the Distributed Cache:

```
bash
Copy code
hdfs dfs -put departments.txt /input/departments.txt
```

Step 5: Run the MapReduce Job

Now, run the MapReduce job, passing the input file, output directory, and the distributed cache file.

```
bash
```

```
Copy code
hadoop jar mapsidejoin.jar MapSideJoinDriver /input/employees.txt /output
/input/departments.txt
```

Step 6: Output

The output will be written to the specified HDFS output directory, and it will look something like this:

```
Copy code
101     John,Engineering
102     Alice,HR
103     Bob,Engineering
104     Eve,Sales
```

8.Implement a Reduce-Side Join for two large datasets using MapReduce. Compare the performance of the Reduce-Side join with the Map-Side join.

8solProblem Scenario

Let's consider two large datasets:

1. **Employees dataset (`employees.txt`)**: Contains employee information with department IDs.

   ```
   Copy code
   101,John,1
   102,Alice,2
   103,Bob,1
   104,Eve,3
   ```

2. **Departments dataset (`departments.txt`)**: Contains department information.

   ```
   Copy code
   1,Engineering
   2,HR
   3,Sales
   ```

We want to join these datasets on the **department ID** to produce a dataset that shows the employee name along with their department name.

Step 1: Prepare the Datasets

- **Employees Dataset (`employees.txt`)**:

```
Copy code
101,John,1
102,Alice,2
103,Bob,1
104,Eve,3
```

- **Departments Dataset (`departments.txt`)**:

```
Copy code
1,Engineering
2,HR
3,Sales
```

Step 2: Implement the Reduce-Side Join

In this example, we will implement a **Reduce-Side Join** where the data from both datasets is sent to the reducers. The mapper will tag records from each dataset so the reducer can distinguish between employee and department records.

1. Mapper Class

The mapper will:

- Read both datasets.
- Tag each record with its source (employee or department) and emit the department ID as the key.

```
java
Copy code
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

import java.io.IOException;

public class ReduceSideJoinMapper extends Mapper<LongWritable, Text, Text,
Text> {

    @Override
    protected void map(LongWritable key, Text value, Context context) throws
IOException, InterruptedException {
        String[] fields = value.toString().split(",");

        if (fields.length == 3) {  // Employee record
            String empId = fields[0];
            String empName = fields[1];
            String deptId = fields[2];
            context.write(new Text(deptId), new Text("EMP," + empId + "," +
empName));
        } else if (fields.length == 2) {  // Department record
```

```
            String deptId = fields[0];
            String deptName = fields[1];
            context.write(new Text(deptId), new Text("DEPT," + deptName));
        }
    }
}
```

2. Reducer Class

The reducer will:

- Collect all records with the same department ID.
- Perform the join operation by matching employee and department records.

```
java
Copy code
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

public class ReduceSideJoinReducer extends Reducer<Text, Text, Text, Text> {

    @Override
    protected void reduce(Text key, Iterable<Text> values, Context context)
throws IOException, InterruptedException {
        String departmentName = "";
        List<String> employees = new ArrayList<>();

        // Iterate over all values and separate employee and department data
        for (Text value : values) {
            String[] fields = value.toString().split(",");
            if (fields[0].equals("DEPT")) {
                departmentName = fields[1];  // Get department name
            } else if (fields[0].equals("EMP")) {
                employees.add(fields[1] + "," + fields[2]);  // Collect
employee records
            }
        }

        // Output joined records
        for (String employee : employees) {
            context.write(new Text(employee), new Text(departmentName));
        }
    }
}
```

3. Driver Class

The driver class sets up the job and specifies the mapper and reducer classes.

```java
java
Copy code
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class ReduceSideJoinDriver {
    public static void main(String[] args) throws Exception {
        if (args.length != 3) {
            System.err.println("Usage: ReduceSideJoinDriver <input1 path>
<input2 path> <output path>");
            System.exit(-1);
        }

        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "Reduce-Side Join");

        job.setJarByClass(ReduceSideJoinDriver.class);
        job.setMapperClass(ReduceSideJoinMapper.class);
        job.setReducerClass(ReduceSideJoinReducer.class);

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

        // Set input paths for both datasets
        FileInputFormat.addInputPath(job, new Path(args[0]));  // Employees
dataset
        FileInputFormat.addInputPath(job, new Path(args[1]));  // Departments
dataset
        FileOutputFormat.setOutputPath(job, new Path(args[2]));

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

Step 3: Compile and Package the Program

1. **Compile the Java files**:

```bash
bash
Copy code
javac -cp $(hadoop classpath) -d . *.java
```

2. **Create a JAR file**:

bash
Copy code
```
jar -cvf reducesidejoin.jar -C . .
```

Step 4: Prepare the Input Files

Upload both datasets (`employees.txt` and `departments.txt`) to HDFS.

bash
Copy code
```
hdfs dfs -put employees.txt /input/employees.txt
hdfs dfs -put departments.txt /input/departments.txt
```

Step 5: Run the MapReduce Job

Now, run the MapReduce job with both input paths and an output path.

bash
Copy code
```
hadoop jar reducesidejoin.jar ReduceSideJoinDriver /input/employees.txt
/input/departments.txt /output
```

Step 6: Output

The output will look something like this:

Copy code
```
101,John    Engineering
102,Alice   HR
103,Bob     Engineering
104,Eve     Sales
```

9.Implement a simple Word Count program in Spark using RDDs. 1.Load a text file into an RDD. 2.Count the occurrences of each word in the file. Print the result and save the output in HDFS.

9sol:Step-by-Step Implementation

Step 1: Set Up Your Spark Environment

Make sure you have Spark installed and configured properly on your machine. You can run the Spark program locally or on a cluster.

Step 2: Create a Text File

Create a sample text file (`input.txt`) for testing purposes. You can place this file in your local file system or HDFS.

Sample `input.txt`:

```
csharp
Copy code
Hello World
Hello Spark
Spark is great
```

Step 3: Write the Word Count Program

Here's a simple Spark application that counts the occurrences of each word in a text file using RDDs.

WordCount.scala:

```
scala
Copy code
import org.apache.spark.{SparkConf, SparkContext}

object WordCount {
  def main(args: Array[String]): Unit = {
    // Check if the user provided input and output paths
    if (args.length != 2) {
      println("Usage: WordCount <input_path> <output_path>")
      System.exit(1)
    }

    // Set up the Spark configuration and context
    val conf = new SparkConf().setAppName("WordCount").setMaster("local[*]")
    val sc = new SparkContext(conf)

    // Load the text file into an RDD
    val inputPath = args(0)
    val outputPath = args(1)

    val textFile = sc.textFile(inputPath)

    // Count occurrences of each word
    val wordCounts = textFile
      .flatMap(line => line.split(" ")) // Split each line into words
      .map(word => (word, 1)) // Map each word to a tuple (word, 1)
      .reduceByKey(_ + _) // Reduce by key (word) to count occurrences

    // Print the result
    wordCounts.collect().foreach(println)

    // Save the output in HDFS
```

```
    wordCounts.saveAsTextFile(outputPath)

    // Stop the Spark context
    sc.stop()
  }
}
```

Step 4: Compile and Package the Application

To run the Scala application, you need to compile it and package it into a JAR file.

1. **Create a build.sbt file:**

   sbt
   Copy code
   ```
   name := "WordCount"

   version := "0.1"

   scalaVersion := "2.12.10"

   libraryDependencies += "org.apache.spark" %% "spark-core" % "3.0.1" % "provided"
   ```

2. **Compile the application:**

   bash
   Copy code
   ```
   sbt package
   ```

This will create a JAR file in the `target/scala-2.12/` directory (e.g., `target/scala-2.12/wordcount_2.12-0.1.jar`).

Step 5: Run the Application

1. **Upload your input file to HDFS (if needed):**

   bash
   Copy code
   ```
   hdfs dfs -put input.txt /user/yourusername/input.txt
   ```

2. **Run the Spark application:**

   bash
   Copy code
   ```
   spark-submit --class WordCount target/scala-2.12/wordcount_2.12-0.1.jar /user/yourusername/input.txt /user/yourusername/outputStep 6: Check the Output
   ```

After running the application, you can check the output saved in HDFS.

```bash
hdfs dfs -ls /user/yourusername/output
hdfs dfs -cat /user/yourusername/output/part-00000
```

You should see the word counts displayed in the terminal. The output will look like this:

```scss
(Spark,1)
(Hello,2)
(World,1)
(is,1)
(great,1)
```

Conclusion

You've successfully implemented a simple Word Count program in Spark using RDDs. This program reads a text file, counts the occurrences of each word, prints the result to the console, and saves the output in HDFS.

Feel free to modify the input text file and test the program with different datasets!

You said:

Perform the following RDD operations: 1.Create an RDD from a text file. 2.Apply the map, filter, and flatMap transformations. 3.Use the reduce action to aggregate results. Create two RDDs and perform a join operation. Save the result in HDFS.

ChatGPT said:

Below are step-by-step instructions to perform RDD operations in Apache Spark, including creating RDDs, applying transformations (map, filter, flatMap), using reduce action, and performing a join operation.

Step-by-Step Implementation

Step 1: Set Up Your Spark Environment

Ensure you have Spark installed and configured correctly on your local machine or cluster.

Step 2: Create Sample Input Files

Create two sample text files for this example:

1. **Sample file 1 (`input1.txt`)**:

   ```
   1,John
   2,Alice
   ```

```
3,Bob
4,Eve
```

2. **Sample file 2 (`input2.txt`)**:

```
Copy code
1,1000
2,1500
3,2000
4,3000
```

You can place these files in your local filesystem or HDFS.

Step 3: Write the Spark Application

Here's a simple Spark application that demonstrates the required RDD operations.

RDDOperations.scala:

```
scala
Copy code
import org.apache.spark.{SparkConf, SparkContext}

object RDDOperations {
  def main(args: Array[String]): Unit = {
    // Check if the user provided input and output paths
    if (args.length != 2) {
      println("Usage: RDDOperations <output_path>")
      System.exit(1)
    }

    // Set up the Spark configuration and context
    val conf = new
SparkConf().setAppName("RDDOperations").setMaster("local[*]")
    val sc = new SparkContext(conf)

    // Step 1: Create an RDD from a text file
    val rdd1 = sc.textFile(args(0)) // Path to the first input file
    val rdd2 = sc.textFile(args(1)) // Path to the second input file

    // Step 2: Apply map, filter, and flatMap transformations
    // Example: Extract names from rdd1
    val namesRDD = rdd1.map(line => line.split(",")(1)) // Map to names
    val filteredNamesRDD = namesRDD.filter(name => name.startsWith("A")) //
Filter names starting with 'A'
    val flatMappedRDD = rdd1.flatMap(line => line.split(",")) // FlatMap
example: split each line into individual elements

    // Step 3: Use the reduce action to aggregate results
    val nameCount = namesRDD.count() // Count of names
    println(s"Total names: $nameCount")
```

```
    // Create two RDDs for joining
    val rdd1KeyValue = rdd1.map(line => {
      val parts = line.split(",")
      (parts(0).toInt, parts(1)) // (id, name)
    })

    val rdd2KeyValue = rdd2.map(line => {
      val parts = line.split(",")
      (parts(0).toInt, parts(1).toInt) // (id, salary)
    })

    // Perform join operation
    val joinedRDD = rdd1KeyValue.join(rdd2KeyValue) // Join on the key (id)

    // Step 4: Save the result in HDFS
    joinedRDD.saveAsTextFile("hdfs://<your-hdfs-path>/output") // Specify your
HDFS output path

    // Stop the Spark context
    sc.stop()
  }
}
```

Step 4: Compile and Package the Application

To run the Scala application, you need to compile it and package it into a JAR file.

1. **Create a build.sbt file:**

   ```
   sbt
   Copy code
   name := "RDDOperations"

   version := "0.1"

   scalaVersion := "2.12.10"

   libraryDependencies += "org.apache.spark" %% "spark-core" % "3.0.1" %
   "provided"
   ```

2. **Compile the application:**

   ```
   bash
   Copy code
   sbt package
   ```

This will create a JAR file in the `target/scala-2.12/` directory (e.g., `target/scala-2.12/rddoperations_2.12-0.1.jar`).

Step 5: Run the Application

1. **Upload your input files to HDFS (if needed):**

   bash
   Copy code
   ```
   hdfs dfs -put input1.txt /user/yourusername/input1.txt
   hdfs dfs -put input2.txt /user/yourusername/input2.txt
   ```

2. **Run the Spark application:**

   bash
   Copy code
   ```
   spark-submit --class RDDOperations target/scala-2.12/rddoperations_2.12-0.1.jar /user/yourusername/input1.txt /user/yourusername/input2.txt
   ```

Step 6: Check the Output

After running the application, you can check the output saved in HDFS.

bash
Copy code
```
hdfs dfs -ls hdfs://<your-hdfs-path>/output
hdfs dfs -cat hdfs://<your-hdfs-path>/output/part-00000
```

The output will look something like this:

scss
Copy code
```
(1, (John, 1000))
(2, (Alice, 1500))
(3, (Bob, 2000))
(4, (Eve, 3000))
```

10.Implement matrix multiplication using RDDs in Spark. Multiply two large matrices and store the result in HDFS.

Step-by-Step Implementation

Step 1: Set Up Your Spark Environment

Make sure you have Spark installed and configured properly on your machine or cluster.

Step 2: Create Sample Input Files for Matrices

For this example, let’s assume we have two matrices, **Matrix A** and **Matrix B**, stored in two separate text files.

1. **Sample file for Matrix A (`matrixA.txt`)**:

```
Copy code
1,2,3
4,5,6
7,8,9
```

2. **Sample file for Matrix B (`matrixB.txt`)**:

```
Copy code
9,8,7
6,5,4
3,2,1
```

You can place these files in your local filesystem or HDFS.

Step 3: Write the Spark Application for Matrix Multiplication

Here's a Spark application that reads two matrices from text files, performs matrix multiplication, and saves the result to HDFS.

MatrixMultiplication.scala:

```
scala
Copy code
import org.apache.spark.{SparkConf, SparkContext}

object MatrixMultiplication {
  def main(args: Array[String]): Unit = {
    // Check if the user provided input paths
    if (args.length != 3) {
      println("Usage: MatrixMultiplication <matrixA_path> <matrixB_path>
<output_path>")
      System.exit(1)
    }

    // Set up the Spark configuration and context
    val conf = new
SparkConf().setAppName("MatrixMultiplication").setMaster("local[*]")
    val sc = new SparkContext(conf)

    // Step 1: Load matrices from text files
    val matrixA = sc.textFile(args(0)).map(line =>
line.split(",").map(_.toDouble))
    val matrixB = sc.textFile(args(1)).map(line =>
line.split(",").map(_.toDouble))

    // Step 2: Prepare matrix B for multiplication
    // Transpose matrix B for easier multiplication
    val transposedB = matrixB.zipWithIndex.flatMap { case (row, index) =>
      row.indices.map(j => ((j, index), row(j)))
```

```scala
    }

    // Step 3: Perform the multiplication
    val result = matrixA.zipWithIndex.flatMap { case (rowA, i) =>
      transposedB.collect {
        case ((j, k), valueB) if i == k =>
          ((i, j), rowA.zipWithIndex.map { case (valueA, _) => valueA *
valueB }.sum)
      }
    }

    // Step 4: Convert result to a format suitable for saving
    val resultMatrix = result.reduceByKey(_ + _)

    // Step 5: Save the result in HDFS
    resultMatrix
      .map { case ((i, j), value) => s"$i,$j,$value" }
      .saveAsTextFile(args(2)) // Specify your HDFS output path

    // Stop the Spark context
    sc.stop()
  }
}
```

Step 4: Compile and Package the Application

To run the Scala application, you need to compile it and package it into a JAR file.

1. **Create a build.sbt file:**

   ```
   sbt
   Copy code
   name := "MatrixMultiplication"

   version := "0.1"

   scalaVersion := "2.12.10"

   libraryDependencies += "org.apache.spark" %% "spark-core" % "3.0.1" %
   "provided"
   ```

2. **Compile the application:**

   ```
   bash
   Copy code
   sbt package
   ```

This will create a JAR file in the `target/scala-2.12/` directory (e.g., `target/scala-2.12/matrixmultiplication_2.12-0.1.jar`).

Step 5: Run the Application

1. **Upload your input files to HDFS (if needed):**

   ```bash
   hdfs dfs -put matrixA.txt /user/yourusername/matrixA.txt
   hdfs dfs -put matrixB.txt /user/yourusername/matrixB.txt
   ```

2. **Run the Spark application:**

   ```bash
   spark-submit --class MatrixMultiplication
   target/scala-2.12/matrixmultiplication_2.12-0.1.jar
   /user/yourusername/matrixA.txt /user/yourusername/matrixB.txt
   /user/yourusername/output
   ```

Step 6: Check the Output

After running the application, you can check the output saved in HDFS.

```bash
hdfs dfs -ls /user/yourusername/output
hdfs dfs -cat /user/yourusername/output/part-00000
```

The output will look something like this:

```scss
(0,0,30.0)
(0,1,24.0)
(0,2,18.0)
(1,0,84.0)
(1,1,69.0)
(1,2,54.0)
(2,0,138.0)
(2,1,114.0)
(2,2,90.0)
```