Task-3: MapReduce Program

Install Eclipse in linux.

- ***Install snap: apt install snapd -y***
- Create a symbolic link between /var/lib/snapd/snap and /snap. Doing so gives Snap apps access to your home directory after the installation: ***ln -s /var/lib/snapd/snap /snap***
- Now, update the Snap package list by running the below command: ***snap refresh***
- Run the following command to install the latest stable version of Eclipse. This command downloads the latest stable version of Eclipse from the Snap store and installs it on your system: ***snap install --classic eclipse***

The entire MapReduce program can be fundamentally divided into three parts:

Create a folder named wordcount in eclipse and keep 3 java files into it.

- Mapper Phase Code: map.java
- Reducer Phase Code: reduce.java
- Driver Code: driver.java

1. **Mapper Class:**

- Create a class Map that extends the class Mapper which is already defined in the MapReduce Framework.
- We define the data types of input and output key/value pair after the class declaration using angle brackets.
- Both the input and output of the Mapper is a key/value pair.
- Input:
    - The *key* is nothing but the offset of each line in the text file: *LongWritable*
    - The *value* is each individual line (as shown in the figure at the right): *Text*
- Output:
    - The *key* is the tokenized words: *Text*
    - We have the hardcoded *value* in our case which is 1: *IntWritable*
    - Example – Dear 1, Bear 1, etc.
- Following is the java code where we have tokenized each word and assigned them a hardcoded value equal to *1*.
- Create a java named ***map.java*** and copy the following content into it.

```
public static class Map extends Mapper&lt;LongWritable,Text,Text,IntWritable&gt;
{
     public void map(LongWritable key, Text value, Context context) throws
IOException,InterruptedException {
          String line = value.toString();
          StringTokenizer tokenizer = new StringTokenizer(line);
          while (tokenizer.hasMoreTokens()) {
               value.set(tokenizer.nextToken());
               context.write(value, new IntWritable(1));
           }
```

2. Reducer Class:

- Create a class Reduce which extends class Reducer like that of Mapper.
- We define the data types of input and output key/value pair after the class declaration using angle brackets as done for Mapper.
- Both the input and the output of the Reducer is a key-value pair.
- Input:
    - The *key* nothing but those unique words which have been generated after the sorting and shuffling phase: *Text*
    - The *value* is a list of integers corresponding to each key: *IntWritable*
    - Example – Bear, [1, 1], etc.
- Output:
    - The *key* is all the unique words present in the input text file: *Text*
    - The *value* is the number of occurrences of each of the unique words: *IntWritable*
    - Example – Bear, 2; Car, 3, etc.
- Here, we have aggregated the values present in each of the list corresponding to each key and produced the final answer.
- In general, a single reducer is created for each of the unique words, but, we can specify the number of reducer in mapred-site.xml.
- Create a java file named ***reduce.java*** and copy the following content into it.

```
public static class Reduce extends
Reducer&lt;Text,IntWritable,Text,IntWritable&gt; {
     public void reduce(Text key, Iterable&lt;IntWritable&gt; values,Context
context)
           throws IOException,InterruptedException {
                int sum=0;
                for(IntWritable x: values)
                {
                      sum+=x.get();
                }
                context.write(key, new IntWritable(sum));
           }
      }
```

3. Driver Code:

- In the driver class, we set the configuration of our MapReduce job to run in Hadoop.
- We specify the name of the job, the data type of input/output of the mapper and reducer.
- We also specify the names of the mapper and reducer classes.
- The path of the input and output folder is also specified.
- The method setInputFormatClass () is used for specifying how a Mapper will read the input data or what will be the unit of work. Here, we have chosen TextInputFormat so that a single line is read by the mapper at a time from the input text file.
- The main () method is the entry point for the driver. In this method, we instantiate a new Configuration object for the job.
- Create a java named ***driver.java*** and copy the following content into it.

```
package co.edureka.mapreduce;
import java.io.IOException;
import java.util.StringTokenizer;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.fs.Path;

public class WordCount{
    public static class Map extends
Mapper&amp;lt;LongWritable,Text,Text,IntWritable&amp;gt; {
        public void map(LongWritable key, Text value,Context context) throws
IOException,InterruptedException{
            String line = value.toString();
            StringTokenizer tokenizer = new StringTokenizer(line);
            while (tokenizer.hasMoreTokens()) {
                value.set(tokenizer.nextToken());
                context.write(value, new IntWritable(1));
            }
        }
    }
```

```
    public static class Reduce extends
Reducer&amp;lt;Text,IntWritable,Text,IntWritable&amp;gt; {
        public void reduce(Text key, Iterable&amp;lt;IntWritable&amp;gt;
values,Context context) throws IOException,InterruptedException {
            int sum=0;
            for(IntWritable x: values)
            {
                sum+=x.get();
            }
            context.write(key, new IntWritable(sum));
        }
    }
    public static void main(String[] args) throws Exception {
        Configuration conf= new Configuration();
        Job job = new Job(conf,"My Word Count Program");
        job.setJarByClass(WordCount.class);
        job.setMapperClass(Map.class);
        job.setReducerClass(Reduce.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        job.setInputFormatClass(TextInputFormat.class);
        job.setOutputFormatClass(TextOutputFormat.class);
        Path outputPath = new Path(args[1]);
        //Configuring the input/output path from the filesystem into the job
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        //deleting the output path automatically from hdfs so that we don't have
to delete it explicitly
        outputPath.getFileSystem(conf).delete(outputPath);
        //exiting the job only if the flag value becomes false
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

4. **Export and Run the MapReduce code:**

Export the project and create a jar file named ***hadoop-mapreduce-example.jar.***

Then run the command for running a MapReduce code is:

```
hadoop jar hadoop-mapreduce-example.jar WordCount /sample/input
/sample/output
```