BIG DATA ANALYSIS - ASSIGNMENT 01 - GROUP NO : 23

Q. no:11 -

- Similar to Q03, but streaming interface. Find the numbers with maximum and minimum number of occurrences using streaming interface.
- Input
  - i/p 1: number of mapped instances.
  - i/p 2: streaming data 1M+ numbers (e.g. cat file).
- Output
  - o/p 1: number with maximum occurrences and occurrences count.
  - o/p 2: number with minimum occurrences and occurrences count.

Basic understanding -

Lets take basic Example

- Content of input text file:
  - Hello World Bye World
  - Hello Hadoop Goodbye Hadoop

1. **The input is fed to the mapper, and each line is given to separate mappers.**

- Output of Mapper - 1:
  - <Hello, 1>
  - <World, 1>
  - <Bye, 1>
  - <World, 1>

- Output of Mapper - 2:
  - <Hello, 1>
  - <Hadoop,1>
  - <Goodbye, 1>
  - <Hadoop, 1>

2. **The output of mappers is given to combiners as input.**

- Output of Combiner -1:
    - <Bye, 1>
    - <Hello, 1>
    - <World, 2>

- Output of Combiner -2:
    - <Goodbye, 1>
    - <Hadoop, 2>
    - <Hello, 1>

3. **The output of combiners is arranged in sorted order. These output become input to shuffle-step. The output of the combiner is grouped with respect to the keys.** (e.g. Bye, Goodbye, Hadoop, Hello, World)

- Output of Shuffle step:
    - <Bye, 1>
    - <Goodbye, 1>
    - <Hadoop, 2>
    - <Hello, 1>
    - <Hello, 1>
    - <World, 2>

4. **These outputs are fed to individual reducers. The reducer adds all the values with same key.**

- Output of Reducer:
    - <Bye,1>
    - <Goodbye, 1>
    - <Hadoop, 2>
    - <Hello, 2>
    - <World, 2>

Source Code -

- **Header Files Required:**

To perform few functions (such as configuration, to divide string into tokens, to use integer variables, to use string data, to create new job, mapper class, reducer class, and to recognize file format for input and output) we make use of certain header file:

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

- Class WordCount:

1. Name of our class is WordCount. Under this class we write two functions - mapper and reducer

   public class WordCount

2. First we extend the mapper class TokenizerMapper. The 4 arguments stands for <key-in, value-in, key-out, value-out>.
   Object - address of text-file where input data is present.
   Text - the data present in file.
   Text - Word (or key).
   IntWritable - value of each key.

public static class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable>

3. Then we create a new word “word” which is of Text type, and one more variable “one” of type IntWritable with value 1.

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

4. Now we write the MAP function. It has 3 arguments.
Object key - Input (the address where data is present).
Text value - value of data in that address.
Context context - used to emit output to the next stage.

public void map(Object key, Text value, Context context) throws IOException, InterruptedException

5. The data must be first converted to string and then tokenized. For this we imported StringTokenizer which will divide string to token and put in variable “itr”.

StringTokenizer itr = new StringTokenizer(value.toString());
while (itr.hasMoreTokens())

6. Then each token is taken and put to “word” variable until no more tokens remain in “itr”. The output will be (word, one) format.

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

7. Next we write the reducer function. We extend the reducer class “IntSumReducer”.It has 4 arguments-
Text - Word / key (key-in).

IntWritable - Value (value-in).
Text - Same word (key-out).
IntWritable - Value after adding the values with same word (value-out).

public static class IntSumReducer extends
Reducer<Text,IntWritable,Text,IntWritable>

8. Then we declare a variable “result” of type IntWritable which will store how many times key is appearing.

   private IntWritable result = new IntWritable();

9. Then we write a reduce function which has 3 arguments.
   Text key - Input Key.
   Iterable<IntWritable> values - Value corresponding to key.
   Context context - Used to emit the output.

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

10. Initially sum is 0 for a key, and we iterate for every value of a particular key and add all the values. Then we put the sum into our result and output it using context in the form <key, value> pair.

```
int sum = 0;
for (IntWritable val : values) {
sum += val.get();
}
result.set(sum);
context.write(key, result);
```

- Main Function:

1. We create a configuration object “conf”.

   Configuration conf = new Configuration();

2. Using the configuration object we create a new instance of job “job”.

   Job job = Job.getInstance(conf, "word count");

3. Then we set the class name which is “WordCount” to jar.

   job.setJarByClass(WordCount.class);

4. Then set the mapper class name “TokenizerMapper” to Mapperclass .

   j**ob.setMapperClass(TokenizerMapper.class);**

5. Then set the reducer class name “IntSumReducer” to CombinerClass and ReducerClass.

   job.setCombinerClass(IntSumReducer.class);
   job.setReducerClass(IntSumReducer.class);

6. We set OutputKeyClass to “Text” and OutputValueClass to “Intwritable”.

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

7. The input file path is set by using InputPath to FileInputFormat.

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

8. Then we also say where output should be stored.

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

9. Then we wait for everything to complete and if everything works fine it will return “true” and exit successfully , otherwise System.exit(1) error will be shown.

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

Steps to Compile and run the code:

- **export HADOOP_CLASSPATH=$(hadoop classpath)**

To get a classpath for hadoop.

- **mkdir wordcount_classes**
- **javac -cp $HADOOP_CLASSPATH -d wordcount_classes WordCount.java**

Create directory wordcount_classes and compile file WordCount.java

- **jar -cvf wordcount.jar -C wordcount_classes/ .**

Create jar file.

- **hdfs dfs -mkdir -p /user/hadoopusr/Desktop/WordCountTutorial/Input**
- **hdfs dfs -put WordCount.txt /user/hadoopusr/Desktop/WordCount/Input**

We make a directory “Input” and give input from WordCount.txt.

- **hadoop jar wordcount.jar WordCount /user/hadoopusr/Desktop/WordCount/Output**

Then we run the program.

- **hdfs dfs -ls /user/hadoopusr/Desktop/WordCount/Output**

Then we check for the output from hadoop job.