MAPREDUCE - COMBINERS

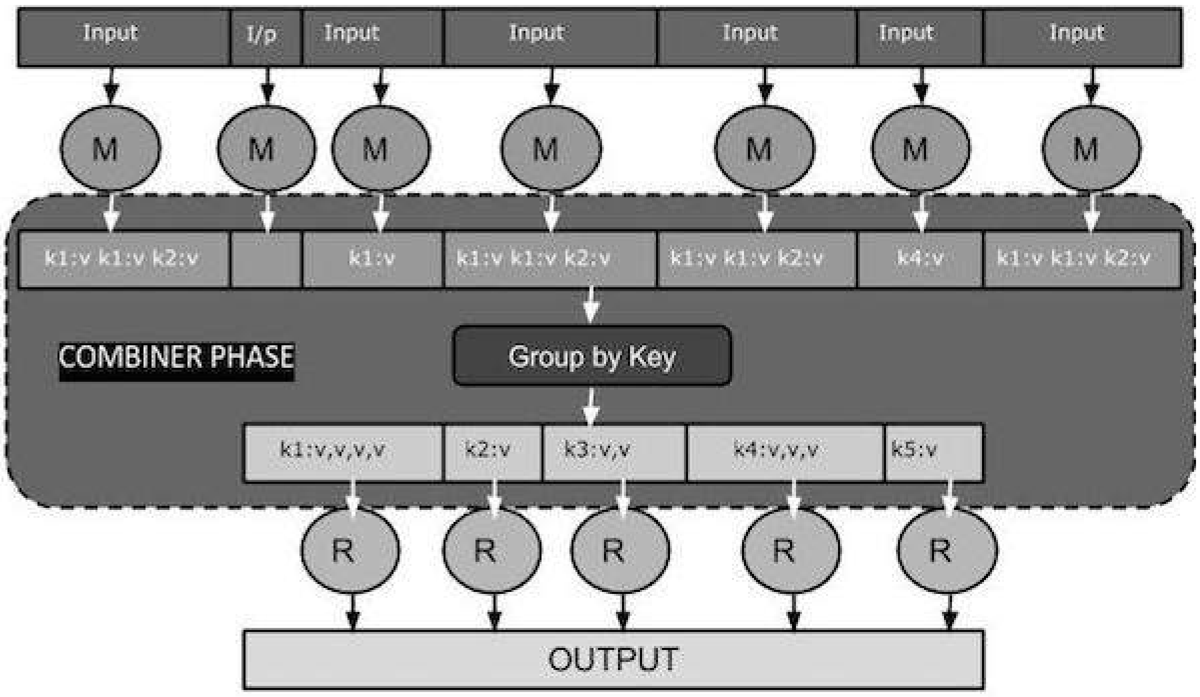
A Combiner, also known as a **semi-reducer,** is an optional class that operates by accepting the inputs from the Map class and thereafter passing the output key-value pairs to the Reducer class.

The main function of a Combiner is to summarize the map output records with the same key. The output $key - valuecollection$ of the combiner will be sent over the network to the actual Reducer task as input.

Combiner

The Combiner class is used in between the Map class and the Reduce class to reduce the volume of data transfer between Map and Reduce. Usually, the output of the map task is large and the data transferred to the reduce task is high.

The following MapReduce task diagram shows the COMBINER PHASE.



How Combiner Works?

Here is a brief summary on how MapReduce Combiner works −

- A combiner does not have a predefined interface and it must implement the Reducer interface’s reduce method.
- A combiner operates on each map output key. It must have the same output key-value types as the Reducer class.
- A combiner can produce summary information from a large dataset because it replaces the original Map output.

Although, Combiner is optional yet it helps segregating data into multiple groups for Reduce phase, which makes it easier to process.

MapReduce Combiner Implementation

The following example provides a theoretical idea about combiners. Let us assume we have the following input text file named **input.txt** for MapReduce.

```
What do you mean by Object
What do you know about Java
What is Java Virtual Machine
How Java enabled High Performance
```

The important phases of the MapReduce program with Combiner are discussed below.

Record Reader

This is the first phase of MapReduce where the Record Reader reads every line from the input text file as text and yields output as key-value pairs.

Input − Line by line text from the input file.

Output − Forms the key-value pairs. The following is the set of expected key-value pairs.

```
<1, What do you mean by Object>
<2, What do you know about Java>
<3, What is Java Virtual Machine>
<4, How Java enabled High Performance>
```

Map Phase

The Map phase takes input from the Record Reader, processes it, and produces the output as another set of key-value pairs.

Input − The following key-value pair is the input taken from the Record Reader.

```
<1, What do you mean by Object>
<2, What do you know about Java>
<3, What is Java Virtual Machine>
<4, How Java enabled High Performance>
```

The Map phase reads each key-value pair, divides each word from the value using StringTokenizer, treats each word as key and the count of that word as value. The following code snippet shows the Mapper class and the map function.

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

   public void map(Object key, Text value, Context context) throws IOException,
InterruptedException
   {
      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens())
      {
         word.set(itr.nextToken());
         context.write(word, one);
      }
   }
}
```

Output − The expected output is as follows −

```
<What,1> <do,1> <you,1> <mean,1> <by,1> <Object,1>
<What,1> <do,1> <you,1> <know,1> <about,1> <Java,1>
<What,1> <is,1> <Java,1> <Virtual,1> <Machine,1>
<How,1> <Java,1> <enabled,1> <High,1> <Performance,1>
```

Combiner Phase

The Combiner phase takes each key-value pair from the Map phase, processes it, and produces the output as **key-value collection** pairs.

Input − The following key-value pair is the input taken from the Map phase.

```
<What,1> <do,1> <you,1> <mean,1> <by,1> <Object,1>
<What,1> <do,1> <you,1> <know,1> <about,1> <Java,1>
<What,1> <is,1> <Java,1> <Virtual,1> <Machine,1>
<How,1> <Java,1> <enabled,1> <High,1> <Performance,1>
```

The Combiner phase reads each key-value pair, combines the common words as key and values as collection. Usually, the code and operation for a Combiner is similar to that of a Reducer. Following is the code snippet for Mapper, Combiner and Reducer class declaration.

```
job.setMapperClass(TokenizerMapper.class);
job.setCombinerClass(IntSumReducer.class);
job.setReducerClass(IntSumReducer.class);
```

Output − The expected output is as follows −

```
<What,1,1,1> <do,1,1> <you,1,1> <mean,1> <by,1> <Object,1>
<know,1> <about,1> <Java,1,1,1>
<is,1> <Virtual,1> <Machine,1>
<How,1> <enabled,1> <High,1> <Performance,1>
```

Reducer Phase

The Reducer phase takes each key-value collection pair from the Combiner phase, processes it, and passes the output as key-value pairs. Note that the Combiner functionality is same as the Reducer.

Input − The following key-value pair is the input taken from the Combiner phase.

```
<What,1,1,1> <do,1,1> <you,1,1> <mean,1> <by,1> <Object,1>
<know,1> <about,1> <Java,1,1,1>
<is,1> <Virtual,1> <Machine,1>
<How,1> <enabled,1> <High,1> <Performance,1>
```

The Reducer phase reads each key-value pair. Following is the code snippet for the Combiner.

```
public static class IntSumReducer extends Reducer<Text,IntWritable,Text,IntWritable>
{
   private IntWritable result = new IntWritable();

   public void reduce(Text key, Iterable<IntWritable> values,Context context) throws
IOException, InterruptedException
   {
      int sum = 0;
      for (IntWritable val : values)
      {
         sum += val.get();
      }
      result.set(sum);
      context.write(key, result);
   }
}
```

Output − The expected output from the Reducer phase is as follows −

```
<What,3> <do,2> <you,2> <mean,1> <by,1> <Object,1>
<know,1> <about,1> <Java,3>
<is,1> <Virtual,1> <Machine,1>
<How,1> <enabled,1> <High,1> <Performance,1>
```

Record Writer

This is the last phase of MapReduce where the Record Writer writes every key-value pair from the Reducer phase and sends the output as text.

Input − Each key-value pair from the Reducer phase along with the Output format.

Output − It gives you the key-value pairs in text format. Following is the expected output.

```
What           3
do             2
you            2
mean           1
by             1
Object         1
know           1
about          1
Java           3
is             1
Virtual        1
Machine        1
How            1
enabled        1
High           1
Performance    1
```

Example Program

The following code block counts the number of words in a program.

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

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

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

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

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

      public void map(Object key, Text value, Context context) throws IOException,
InterruptedException
      {
         StringTokenizer itr = new StringTokenizer(value.toString());
         while (itr.hasMoreTokens())
         {
            word.set(itr.nextToken());
            context.write(word, one);
         }
      }
   }

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

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

      job.setJarByClass(WordCount.class);
      job.setMapperClass(TokenizerMapper.class);
      job.setCombinerClass(IntSumReducer.class);
      job.setReducerClass(IntSumReducer.class);

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

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

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

Save the above program as **WordCount.java**. The compilation and execution of the program is given below.

Compilation and Execution

Let us assume we are in the home directory of Hadoop user *forexample, /home/hadoop*.

Follow the steps given below to compile and execute the above program.

Step 1 − Use the following command to create a directory to store the compiled java classes.

```
$ mkdir units
```

Step 2 − Download Hadoop-core-1.2.1.jar, which is used to compile and execute the MapReduce program. You can download the jar from mvnrepository.com.

Let us assume the downloaded folder is /home/hadoop/.

Step 3 − Use the following commands to compile the **WordCount.java** program and to create a jar for the program.

```
$ javac -classpath hadoop-core-1.2.1.jar -d units WordCount.java
$ jar -cvf units.jar -C units/ .
```

Step 4 − Use the following command to create an input directory in HDFS.

```
$HADOOP_HOME/bin/hadoop fs -mkdir input_dir
```

Step 5 − Use the following command to copy the input file named **input.txt** in the input directory of HDFS.

```
$HADOOP_HOME/bin/hadoop fs -put /home/hadoop/input.txt input_dir
```

Step 6 − Use the following command to verify the files in the input directory.

```
$HADOOP_HOME/bin/hadoop fs -ls input_dir/
```

Step 7 − Use the following command to run the Word count application by taking input files from the input directory.

```
$HADOOP_HOME/bin/hadoop jar units.jar hadoop.ProcessUnits input_dir output_dir
```

Wait for a while till the file gets executed. After execution, the output contains a number of input splits, Map tasks, and Reducer tasks.

Step 8 − Use the following command to verify the resultant files in the output folder.

```
$HADOOP_HOME/bin/hadoop fs -ls output_dir/
```

Step 9 − Use the following command to see the output in **Part-00000** file. This file is generated by HDFS.

```
$HADOOP_HOME/bin/hadoop fs -cat output_dir/part-00000
```

Following is the output generated by the MapReduce program.

```
What           3
do              2
you            2
mean            1
by              1
Object         1
know           1
about          1
Java           3
is              1
Virtual        1
Machine        1
How             1
enabled        1
High           1
Performance    1
```

Loading [MathJax]/jax/output/HTML-CSS/jax.js