MAPREDUCE - PARTITIONER

A partitioner works like a condition in processing an input dataset. The partition phase takes place after the Map phase and before the Reduce phase.

The number of partitioners is equal to the number of reducers. That means a partitioner will divide the data according to the number of reducers. Therefore, the data passed from a single partitioner is processed by a single Reducer.

Partitioner

A partitioner partitions the key-value pairs of intermediate Map-outputs. It partitions the data using a user-defined condition, which works like a hash function. The total number of partitions is same as the number of Reducer tasks for the job. Let us take an example to understand how the partitioner works.

MapReduce Partitioner Implementation

For the sake of convenience, let us assume we have a small table called Employee with the following data. We will use this sample data as our input dataset to demonstrate how the partitioner works.

Id	Name	Age	Gender	Salary
1201	gopal	45	Male	50,000
1202	manisha	40	Female	50,000
1203	khalil	34	Male	30,000
1204	prasanth	30	Male	30,000
1205	kiran	20	Male	40,000
1206	laxmi	25	Female	35,000
1207	bhavya	20	Female	15,000
1208	reshma	19	Female	15,000
1209	kranthi	22	Male	22,000
1210	Satish	24	Male	25,000
1211	Krishna	25	Male	25,000
1212	Arshad	28	Male	20,000
1213	lavanya	18	Female	8,000

We have to write an application to process the input dataset to find the highest salaried employee by gender in different age groups $forexample, below20, between21to30, above30$.

Input Data

The above data is saved as **input.txt** in the “/home/hadoop/hadoopPartitioner” directory and given as input.

```
1201       gopal          45          Male        50000
```

1202	manisha	40	Female	51000
1203	khaleel	34	Male	30000
1204	prasanth	30	Male	31000
1205	kiran	20	Male	40000
1206	laxmi	25	Female	35000
1207	bhavya	20	Female	15000
1208	reshma	19	Female	14000
1209	kranthi	22	Male	22000
1210	Satish	24	Male	25000
1211	Krishna	25	Male	26000
1212	Arshad	28	Male	20000
1213	lavanya	18	Female	8000

Based on the given input, following is the algorithmic explanation of the program.

Map Tasks

The map task accepts the key-value pairs as input while we have the text data in a text file. The input for this map task is as follows −

Input − The key would be a pattern such as “any special key &plus; filename &plus; line number” $example: key = @input1$ and the value would be the data in that line $example: value = 1201\backslash tgopal\backslash t45\backslash tMale\backslash t50000$.

Method − The operation of this map task is as follows −

- Read the **value** $record data$, which comes as input value from the argument list in a string.
- Using the split function, separate the gender and store in a string variable.

```
String[] str = value.toString().split("\t", -3);
String gender=str[3];
```

- Send the gender information and the record data **value** as output key-value pair from the map task to the **partition task**.

```
context.write(new Text(gender), new Text(value));
```

- Repeat all the above steps for all the records in the text file.

Output − You will get the gender data and the record data value as key-value pairs.

Partitioner Task

The partitioner task accepts the key-value pairs from the map task as its input. Partition implies dividing the data into segments. According to the given conditional criteria of partitions, the input key-value paired data can be divided into three parts based on the age criteria.

Input − The whole data in a collection of key-value pairs.

key = Gender field value in the record.

value = Whole record data value of that gender.

Method − The process of partition logic runs as follows.

- Read the age field value from the input key-value pair.

```
String[] str = value.toString().split("\t");
int age = Integer.parseInt(str[2]);
```

- Check the age value with the following conditions.
    - Age less than or equal to 20
    - Age Greater than 20 and Less than or equal to 30.
    - Age Greater than 30.

```
if(age<=20)
{
   return 0;
}
else if(age>20 && age<=30)
{
   return 1 % numReduceTasks;
}
else
{
   return 2 % numReduceTasks;
}
```

Output − The whole data of key-value pairs are segmented into three collections of key-value pairs. The Reducer works individually on each collection.

Reduce Tasks

The number of partitioner tasks is equal to the number of reducer tasks. Here we have three partitioner tasks and hence we have three Reducer tasks to be executed.

Input − The Reducer will execute three times with different collection of key-value pairs.

key = gender field value in the record.

value = the whole record data of that gender.

Method − The following logic will be applied on each collection.

- Read the Salary field value of each record.

```
String [] str = val.toString().split("\t", -3);
Note: str[4] have the salary field value.
```

- Check the salary with the max variable. If str[4] is the max salary, then assign str[4] to max, otherwise skip the step.

```
if(Integer.parseInt(str[4])>max)
{
   max=Integer.parseInt(str[4]);
}
```

- Repeat Steps 1 and 2 for each key collection Male & Female are the key collections. After executing these three steps, you will find one max salary from the Male key collection and one max salary from the Female key collection.

```
context.write(new Text(key), new IntWritable(max));
```

Output − Finally, you will get a set of key-value pair data in three collections of different age groups. It contains the max salary from the Male collection and the max salary from the Female collection in each age group respectively.

After executing the Map, the Partitioner, and the Reduce tasks, the three collections of key-value pair data are stored in three different files as the output.

All the three tasks are treated as MapReduce jobs. The following requirements and specifications of these jobs should be specified in the Configurations −

- Job name
- Input and Output formats of keys and values
- Individual classes for Map, Reduce, and Partitioner tasks

```
Configuration conf = getConf();

//Create Job
Job job = new Job(conf, "topsal");
job.setJarByClass(PartitionerExample.class);

// File Input and Output paths
FileInputFormat.setInputPaths(job, new Path(arg[0]));
FileOutputFormat.setOutputPath(job,new Path(arg[1]));

//Set Mapper class and Output format for key-value pair.
job.setMapperClass(MapClass.class);
job.setMapOutputKeyClass(Text.class);
job.setMapOutputValueClass(Text.class);

//set partitioner statement
job.setPartitionerClass(CaderPartitioner.class);

//Set Reducer class and Input/Output format for key-value pair.
job.setReducerClass(ReduceClass.class);

//Number of Reducer tasks.
job.setNumReduceTasks(3);

//Input and Output format for data
job.setInputFormatClass(TextInputFormat.class);
job.setOutputFormatClass(TextOutputFormat.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(Text.class);
```

Example Program

The following program shows how to implement the partitioners for the given criteria in a MapReduce program.

```
package partitionerexample;

import java.io.*;

import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.fs.*;

import org.apache.hadoop.mapreduce.lib.input.*;
import org.apache.hadoop.mapreduce.lib.output.*;

import org.apache.hadoop.util.*;

public class PartitionerExample extends Configured implements Tool
{
   //Map class
   public static class MapClass extends Mapper<LongWritable,Text,Text,Text>
   {
      public void map(LongWritable key, Text value, Context context)
      {
         try{
            String[] str = value.toString().split("\t", -3);
            String gender=str[3];
            context.write(new Text(gender), new Text(value));
         }
         catch(Exception e)
         {
            System.out.println(e.getMessage());
         }
      }
   }

   //Reducer class

   public static class ReduceClass extends Reducer<Text,Text,Text,IntWritable>
   {
      public int max = -1;
      public void reduce(Text key, Iterable <Text> values, Context context) throws
IOException, InterruptedException
      {
         max = -1;

         for (Text val : values)
         {
            String [] str = val.toString().split("\t", -3);
            if(Integer.parseInt(str[4])>max)
            max=Integer.parseInt(str[4]);
         }

         context.write(new Text(key), new IntWritable(max));
      }
   }

   //Partitioner class

   public static class CaderPartitioner extends
   Partitioner < Text, Text >
   {
      @Override
      public int getPartition(Text key, Text value, int numReduceTasks)
      {
         String[] str = value.toString().split("\t");
         int age = Integer.parseInt(str[2]);

         if(numReduceTasks == 0)
         {
            return 0;
         }

         if(age<=20)
         {
            return 0;
         }
         else if(age>20 && age<=30)
         {
            return 1 % numReduceTasks;
         }
         else
         {
            return 2 % numReduceTasks;
         }
      }
   }

   @Override
   public int run(String[] arg) throws Exception
   {
      Configuration conf = getConf();

      Job job = new Job(conf, "topsal");
      job.setJarByClass(PartitionerExample.class);

      FileInputFormat.setInputPaths(job, new Path(arg[0]));
      FileOutputFormat.setOutputPath(job,new Path(arg[1]));

      job.setMapperClass(MapClass.class);

      job.setMapOutputKeyClass(Text.class);
      job.setMapOutputValueClass(Text.class);

      //set partitioner statement

      job.setPartitionerClass(CaderPartitioner.class);
      job.setReducerClass(ReduceClass.class);
      job.setNumReduceTasks(3);
      job.setInputFormatClass(TextInputFormat.class);

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

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

   public static void main(String ar[]) throws Exception
   {
      int res = ToolRunner.run(new Configuration(), new PartitionerExample(),ar);
      System.exit(0);
   }
}
```

Save the above code as **PartitionerExample.java** in “/home/hadoop/hadoopPartitioner”. The compilation and execution of the program is given below.

Compilation and Execution

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

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

Step 1 − 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/hadoopPartitioner”

Step 2 − The following commands are used for compiling the program **PartitionerExample.java** and creating a jar for the program.

```
$ javac -classpath hadoop-core-1.2.1.jar -d ProcessUnits.java
$ jar -cvf PartitionerExample.jar -C .
```

Step 3 − Use the following command to create an input directory in HDFS.

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

Step 4 − 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/hadoopPartitioner/input.txt input_dir
```

Step 5 – Use the following command to verify the files in the input directory.

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

Step 6 – Use the following command to run the Top salary application by taking input files from the input directory.

```
$HADOOP_HOME/bin/hadoop jar PartitionerExample.jar partitionerexample.PartitionerExample
input_dir/input.txt 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.

```
15/02/04 15:19:51 INFO mapreduce.Job: Job job_1423027269044_0021 completed successfully
15/02/04 15:19:52 INFO mapreduce.Job: Counters: 49

File System Counters

   FILE: Number of bytes read=467
   FILE: Number of bytes written=426777
   FILE: Number of read operations=0
   FILE: Number of large read operations=0
   FILE: Number of write operations=0

   HDFS: Number of bytes read=480
   HDFS: Number of bytes written=72
   HDFS: Number of read operations=12
   HDFS: Number of large read operations=0
   HDFS: Number of write operations=6

Job Counters

   Launched map tasks=1
   Launched reduce tasks=3

   Data-local map tasks=1

   Total time spent by all maps in occupied slots (ms)=8212
   Total time spent by all reduces in occupied slots (ms)=59858
   Total time spent by all map tasks (ms)=8212
   Total time spent by all reduce tasks (ms)=59858

   Total vcore-seconds taken by all map tasks=8212
   Total vcore-seconds taken by all reduce tasks=59858

   Total megabyte-seconds taken by all map tasks=8409088
   Total megabyte-seconds taken by all reduce tasks=61294592

Map-Reduce Framework

   Map input records=13
   Map output records=13
   Map output bytes=423
   Map output materialized bytes=467

   Input split bytes=119

   Combine input records=0
   Combine output records=0

   Reduce input groups=6
   Reduce shuffle bytes=467
   Reduce input records=13
   Reduce output records=6

   Spilled Records=26
   Shuffled Maps =3
   Failed Shuffles=0
   Merged Map outputs=3
   GC time elapsed (ms)=224
   CPU time spent (ms)=3690

   Physical memory (bytes) snapshot=553816064
   Virtual memory (bytes) snapshot=3441266688

   Total committed heap usage (bytes)=334102528

Shuffle Errors

   BAD_ID=0
   CONNECTION=0
   IO_ERROR=0

   WRONG_LENGTH=0
   WRONG_MAP=0
   WRONG_REDUCE=0

File Input Format Counters

   Bytes Read=361

File Output Format Counters

   Bytes Written=72
```

Step 7 − Use the following command to verify the resultant files in the output folder.

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

You will find the output in three files because you are using three partitioners and three Reducers in your program.

Step 8 − 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
```

Output in Part-00000

```
Female   15000
Male     40000
```

Use the following command to see the output in **Part-00001** file.

```
$HADOOP_HOME/bin/hadoop fs -cat output_dir/part-00001
```

Output in Part-00001

```
Female   35000
Male    31000
```

Use the following command to see the output in **Part-00002** file.

```
$HADOOP_HOME/bin/hadoop fs -cat output_dir/part-00002
```

Output in Part-00002

```
Female  51000
Male   50000
```

Processing math: 100%