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.

### Map Tasks

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

**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.

### 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.

### 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));

### Example 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);

}

}

## 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.

## 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.

### Map Phase

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

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.

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);

}

}

}

### 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.

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.

job.setMapperClass(TokenizerMapper.class);

job.setCombinerClass(IntSumReducer.class);

job.setReducerClass(IntSumReducer.class);

### 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.

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);

}

}

### Example 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);

}

}

Localhost50070