MAPREDUCE - HADOOP IMPLEMENTATION

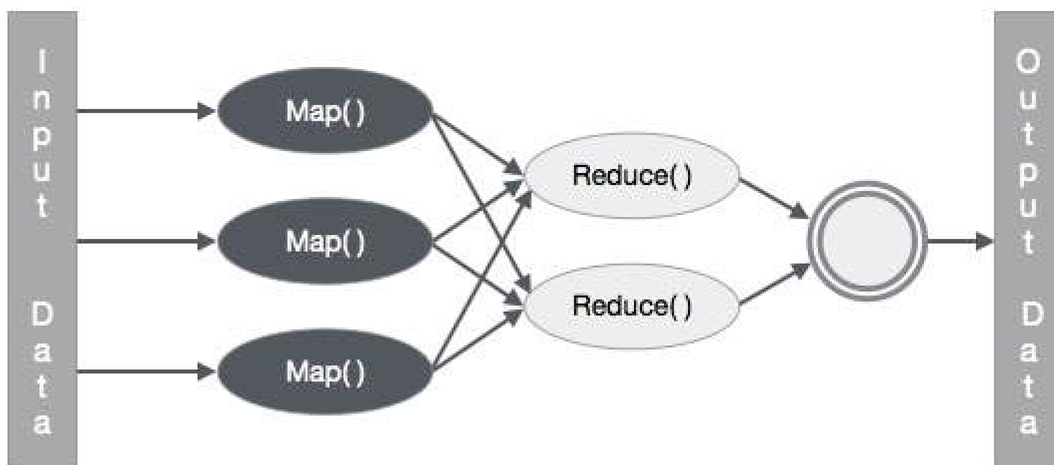
Copyright © tutorialspoint.com

MapReduce is a framework that is used for writing applications to process huge volumes of data on large clusters of commodity hardware in a reliable manner. This chapter takes you through the operation of MapReduce in Hadoop framework using Java.

MapReduce Algorithm

Generally MapReduce paradigm is based on sending map-reduce programs to computers where the actual data resides.

- During a MapReduce job, Hadoop sends Map and Reduce tasks to appropriate servers in the cluster.
- The framework manages all the details of data-passing like issuing tasks, verifying task completion, and copying data around the cluster between the nodes.
- Most of the computing takes place on the nodes with data on local disks that reduces the network traffic.
- After completing a given task, the cluster collects and reduces the data to form an appropriate result, and sends it back to the Hadoop server.



Inputs and Outputs *JavaPerspective*

The MapReduce framework operates on key-value pairs, that is, the framework views the input to the job as a set of key-value pairs and produces a set of key-value pair as the output of the job, conceivably of different types.

The key and value classes have to be serializable by the framework and hence, it is required to implement the Writable interface. Additionally, the key classes have to implement the WritableComparable interface to facilitate sorting by the framework.

Both the input and output format of a MapReduce job are in the form of key-value pairs −

Input <k1, v1> -> map -> <k2, v2>-> reduce -> <k3, v3> *Output*.

	Input	Output
Map	<k1, v1>	list $< k2, v2 >$
Reduce	<k2, list$v2$>	list $< k3, v3 >$

MapReduce Implementation

The following table shows the data regarding the electrical consumption of an organization. The table includes the monthly electrical consumption and the annual average for five consecutive years.

	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec	Avg
1979	23	23	2	43	24	25	26	26	26	26	25	26	25
1980	26	27	28	28	28	30	31	31	31	30	30	30	29
1981	31	32	32	32	33	34	35	36	36	34	34	34	34
1984	39	38	39	39	39	41	42	43	40	39	38	38	40
1985	38	39	39	39	39	41	41	41	00	40	39	39	45

We need to write applications to process the input data in the given table to find the year of maximum usage, the year of minimum usage, and so on. This task is easy for programmers with finite amount of records, as they will simply write the logic to produce the required output, and pass the data to the written application.

Let us now raise the scale of the input data. Assume we have to analyze the electrical consumption of all the large-scale industries of a particular state. When we write applications to process such bulk data,

- They will take a lot of time to execute.
- There will be heavy network traffic when we move data from the source to the network server.

To solve these problems, we have the MapReduce framework.

Input Data

The above data is saved as **sample.txt** and given as input. The input file looks as shown below.

```
1979   23   23   2    43   24   25   26   26   26   26   25   26   25
1980   26   27   28   28   28   30   31   31   31   30   30   30   29
1981   31   32   32   32   33   34   35   36   36   34   34   34   34
1984   39   38   39   39   39   41   42   43   40   39   38   38   40
1985   38   39   39   39   39   41   41   41   00   40   39   39   45
```

Example Program

The following program for the sample data uses MapReduce framework.

```
package hadoop;

import java.util.*;
import java.io.IOException;
import java.io.IOException;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
import org.apache.hadoop.util.*;
```

```
public class ProcessUnits
{
   //Mapper class
   public static class E_EMapper extends MapReduceBase implements
   Mapper<LongWritable,  /*Input key Type */
   Text,                 /*Input value Type*/
   Text,                 /*Output key Type*/
   IntWritable>          /*Output value Type*/
   {
      //Map function
      public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable>
output, Reporter reporter) throws IOException
      {
         String line = value.toString();
         String lasttoken = null;
         StringTokenizer s = new StringTokenizer(line,"\t");
         String year = s.nextToken();

         while(s.hasMoreTokens()){
            lasttoken=s.nextToken();
         }

         int avgprice = Integer.parseInt(lasttoken);
         output.collect(new Text(year), new IntWritable(avgprice));
      }
   }

   //Reducer class

   public static class E_EReduce extends MapReduceBase implements
   Reducer< Text, IntWritable, Text, IntWritable >
   {
      //Reduce function
      public void reduce(Text key, Iterator <IntWritable> values, OutputCollector>Text,
IntWritable> output, Reporter reporter) throws IOException
      {
         int maxavg=30;
         int val=Integer.MIN_VALUE;
         while (values.hasNext())
         {
            if((val=values.next().get())>maxavg)
            {
               output.collect(key, new IntWritable(val));
            }
         }
      }
   }

   //Main function

   public static void main(String args[])throws Exception
   {
      JobConf conf = new JobConf(Eleunits.class);

      conf.setJobName("max_eletricityunits");

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

      conf.setMapperClass(E_EMapper.class);
      conf.setCombinerClass(E_EReduce.class);
      conf.setReducerClass(E_EReduce.class);

      conf.setInputFormat(TextInputFormat.class);
      conf.setOutputFormat(TextOutputFormat.class);

      FileInputFormat.setInputPaths(conf, new Path(args[0]));
      FileOutputFormat.setOutputPath(conf, new Path(args[1]));
```

```
        JobClient.runJob(conf);
    }
}
```

Save the above program into **ProcessUnits.java**. The compilation and execution of the program is given below.

Compilation and Execution of ProcessUnits Program

Let us assume we are in the home directory of Hadoop user *e. g. /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. Download the jar from mvnrepository.com. Let us assume the download folder is /home/hadoop/.

Step 3 − The following commands are used to compile the **ProcessUnits.java** program and to create a jar for the program.

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

Step 4 − The following command is used to create an input directory in HDFS.

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

Step 5 − The following command is used to copy the input file named **sample.txt** in the input directory of HDFS.

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

Step 6 − The following command is used to verify the files in the input directory

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

Step 7 − The following command is used to run the Eleunit_max 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, Reducer tasks, etc.

```
INFO mapreduce.Job: Job job_1414748220717_0002
completed successfully
14/10/31 06:02:52
INFO mapreduce.Job: Counters: 49

File System Counters

    FILE: Number of bytes read=61
    FILE: Number of bytes written=279400
    FILE: Number of read operations=0
    FILE: Number of large read operations=0
    FILE: Number of write operations=0

    HDFS: Number of bytes read=546
    HDFS: Number of bytes written=40
```

```
   HDFS: Number of read operations=9
   HDFS: Number of large read operations=0
   HDFS: Number of write operations=2 Job Counters

   Launched map tasks=2
   Launched reduce tasks=1
   Data-local map tasks=2

   Total time spent by all maps in occupied slots (ms)=146137
   Total time spent by all reduces in occupied slots (ms)=441
   Total time spent by all map tasks (ms)=14613
   Total time spent by all reduce tasks (ms)=44120

   Total vcore-seconds taken by all map tasks=146137
   Total vcore-seconds taken by all reduce tasks=44120

   Total megabyte-seconds taken by all map tasks=149644288
   Total megabyte-seconds taken by all reduce tasks=45178880

Map-Reduce Framework

   Map input records=5

   Map output records=5
   Map output bytes=45
   Map output materialized bytes=67

   Input split bytes=208
   Combine input records=5
   Combine output records=5

   Reduce input groups=5
   Reduce shuffle bytes=6
   Reduce input records=5
   Reduce output records=5

   Spilled Records=10
   Shuffled Maps =2
   Failed Shuffles=0
   Merged Map outputs=2

   GC time elapsed (ms)=948
   CPU time spent (ms)=5160

   Physical memory (bytes) snapshot=47749120
   Virtual memory (bytes) snapshot=2899349504

   Total committed heap usage (bytes)=277684224

File Output Format Counters

   Bytes Written=40
```

Step 8 − The following command is used to verify the resultant files in the output folder.

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

Step 9 − The following command is used 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 −

```
1981   34
```

1984 40

1985 45

Step 10 − The following command is used to copy the output folder from HDFS to the local file system.

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

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