Hadoop - MapReduce

MapReduce is a framework using which we can write applications to process huge amounts of data, in parallel, on large clusters of commodity hardware in a reliable manner.

What is MapReduce?

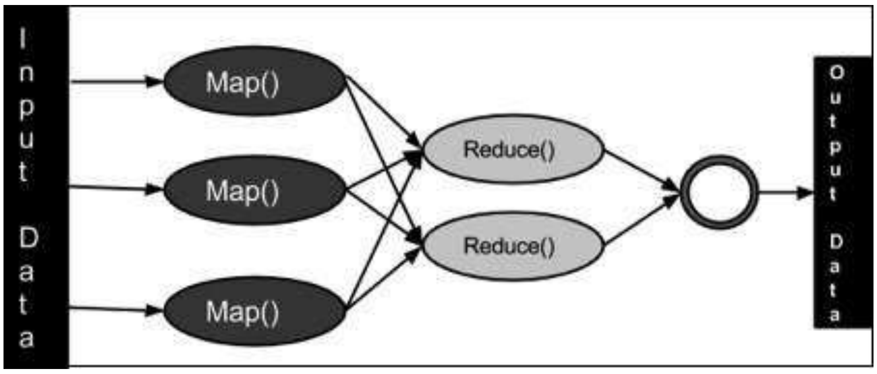
MapReduce is a processing technique and a program model for distributed computing based on java. The MapReduce algorithm contains two important tasks, namely Map and Reduce. Map takes a set of data and converts it into another set of data, where individual elements are broken down into tuples (key/value pairs). Secondly, reduce task, which takes the output from a map as an input and combines those data tuples into a smaller set of tuples. As the sequence of the name MapReduce implies, the reduce task is always performed after the map job.

The major advantage of MapReduce is that it is easy to scale data processing over multiple computing nodes. Under the MapReduce model, the data processing primitives are called mappers and reducers. Decomposing a data processing application into *mappers* and *reducers* is sometimes nontrivial. But, once we write an application in the MapReduce form, scaling the application to run over hundreds, thousands, or even tens of thousands of machines in a cluster is merely a configuration change. This simple scalability is what has attracted many programmers to use the MapReduce model.

The Algorithm

- Generally MapReduce paradigm is based on sending the computer to where the data resides!
- MapReduce program executes in three stages, namely map stage, shuffle stage, and reduce stage.
    - **Map stage −** The map or mapper’s job is to process the input data. Generally the input data is in the form of file or directory and is stored in the Hadoop file system (HDFS). The input file is passed to the mapper function line by line. The mapper processes the data and creates several small chunks of data.
    - **Reduce stage −** This stage is the combination of the **Shuffle** stage and the **Reduce** stage. The Reducer’s job is to process the data that comes from the mapper. After processing, it produces a new set of output, which will be stored in the HDFS.

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



Inputs and Outputs (Java Perspective)

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> pairs as the output of the job, conceivably of different types.

The key and the value classes should be in serialized manner by the framework and hence, need to implement the Writable interface. Additionally, the key classes have to implement the Writable-Comparable interface to facilitate sorting by the framework. Input and Output types of a **MapReduce job −** (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>)

Terminology

- **PayLoad −** Applications implement the Map and the Reduce functions, and form the core of the job.
- **Mapper −** Mapper maps the input key/value pairs to a set of intermediate key/value pair.

NamedNode − Node that manages the Hadoop Distributed File System (HDFS).

- **DataNode −** Node where data is presented in advance before any processing takes place.
- **MasterNode −** Node where JobTracker runs and which accepts job requests from clients.
- **SlaveNode −** Node where Map and Reduce program runs.
- **JobTracker −** Schedules jobs and tracks the assign jobs to Task tracker.
- **Task Tracker −** Tracks the task and reports status to JobTracker.
- **Job −** A program is an execution of a Mapper and Reducer across a dataset.
- **Task −** An execution of a Mapper or a Reducer on a slice of data.
- **Task Attempt −** A particular instance of an attempt to execute a task on a SlaveNode.

Example Scenario

Given below is the data regarding the electrical consumption of an organization. It contains the monthly electrical consumption and the annual average for various 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

If the above data is given as input, we have to write applications to process it and produce results such as finding the year of maximum usage, year of minimum usage, and so on. This is a walkover for the programmers with finite number of records. They will simply write the logic to produce the required output, and pass the data to the application written.

But, think of the data representing the electrical consumption of all the largescale industries of a particular state, since its formation.

When we write applications to process such bulk data,

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

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

Given below is the program to the sample data using MapReduce framework.

```
age hadoop;

rt java.util.*;

rt java.io.IOException;
rt java.io.IOException;

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

ic class ProcessUnits {
/Mapper class
ublic static class E_EMapper extends MapReduceBase
implements apper<LongWritable ,/*Input key Type */
ext,                 /*Input value
Type*/ ext,                 /*Output key
Type*/ ntWritable>         /*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
ublic static class E_EReduce extends MapReduceBase implements Reducer< Text, IntWritab

  //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
ublic static void main(String args[])throws Exception {
  JobConf conf = new JobConf(ProcessUnits.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 as **ProcessUnits.java.** The compilation and execution of the program is explained below.

Compilation and Execution of Process Units Program

Let us assume we are in the home directory of a Hadoop user (e.g. /home/hadoop).

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

Step 1

The following command is 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. Visit the following link mvnrepository.com to download the jar. Let us assume the downloaded folder is **/home/hadoop/.**

Step 3

The following commands are used for compiling the **ProcessUnits.java** program and creating 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 the input files from the input directory.

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

Wait for a while until the file is executed. After execution, as shown below, the output will contain the number of input splits, the number of Map tasks, the number of 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
```

Below 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 for analyzing.

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

Important Commands

All Hadoop commands are invoked by the **$HADOOP_HOME/bin/hadoop** command. Running the Hadoop script without any arguments prints the description for all commands.

Usage − hadoop [--config confdir] COMMAND

The following table lists the options available and their description.

Sr.No.	Option & Description
1	**namenode -format** Formats the DFS filesystem.
2	**secondarynamenode** Runs the DFS secondary namenode.
3	**namenode** Runs the DFS namenode.
4	**datanode** Runs a DFS datanode.
5	**dfsadmin** Runs a DFS admin client.
6	**mradmin** Runs a Map-Reduce admin client.
7	**fsck** Runs a DFS filesystem checking utility.
8	**fs** Runs a generic filesystem user client.
9	**balancer** Runs a cluster balancing utility.
10	**oiv** Applies the offline fsimage viewer to an fsimage.
11	**fetchdt** Fetches a delegation token from the NameNode.

12	**jobtracker** Runs the MapReduce job Tracker node.
13	**pipes** Runs a Pipes job.
14	**tasktracker** Runs a MapReduce task Tracker node.
15	**historyserver** Runs job history servers as a standalone daemon.
16	**job** Manipulates the MapReduce jobs.
17	**queue** Gets information regarding JobQueues.
18	**version** Prints the version.
19	**jar <jar>** Runs a jar file.
20	**distcp <srcurl> <desturl>** Copies file or directories recursively.
21	**distcp2 <srcurl> <desturl>** DistCp version 2.
22	**archive -archiveName NAME -p <parent path> <src>* <dest>** Creates a hadoop archive.
23	

	classpath Prints the class path needed to get the Hadoop jar and the required libraries.
24	**daemonlog** Get/Set the log level for each daemon

How to Interact with MapReduce Jobs

Usage − hadoop job [GENERIC_OPTIONS]

The following are the Generic Options available in a Hadoop job.

Sr.No.	GENERIC_OPTION & Description
1	**-submit <job-file>** Submits the job.
2	**-status <job-id>** Prints the map and reduce completion percentage and all job counters.
3	**-counter <job-id> <group-name> <countername>** Prints the counter value.
4	**-kill <job-id>** Kills the job.
5	**-events <job-id> <fromevent-#> <#-of-events>** Prints the events' details received by jobtracker for the given range.
6	**-history [all] <jobOutputDir> - history < jobOutputDir>** Prints job details, failed and killed tip details. More details about the job such as successful tasks and task attempts made for each task can be viewed by specifying the [all] option.
7	**-list[all]** Displays all jobs. -list displays only jobs which are yet to complete.

8	**-kill-task <task-id>** Kills the task. Killed tasks are NOT counted against failed attempts.
9	**-fail-task <task-id>** Fails the task. Failed tasks are counted against failed attempts.
10	**-set-priority <job-id> <priority>** Changes the priority of the job. Allowed priority values are VERY_HIGH, HIGH, NORMAL, LOW, VERY_LOW

To see the status of job

```
$ $HADOOP_HOME/bin/hadoop  job -status <JOB-ID>
e.g.
$ $HADOOP_HOME/bin/hadoop  job -status job_201310191043_0004
```

To see the history of job output-dir

```
$ $HADOOP_HOME/bin/hadoop  job -history <DIR-NAME>
e.g.
$ $HADOOP_HOME/bin/hadoop  job -history /user/expert/output
```

To kill the job

```
$ $HADOOP_HOME/bin/hadoop  job -kill <JOB-ID>
e.g.
$ $HADOOP_HOME/bin/hadoop  job -kill job_201310191043_0004
```