MAPREDUCE - QUICK GUIDE

MAPREDUCE - INTRODUCTION

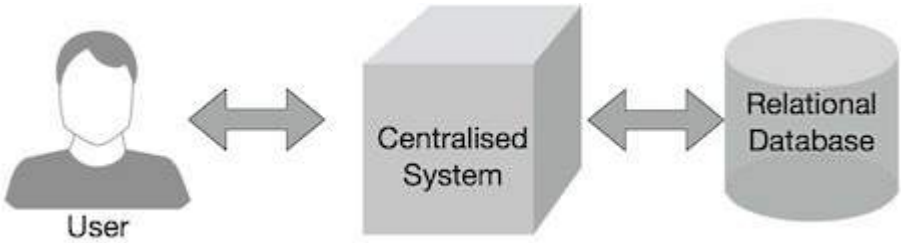
MapReduce is a programming model for writing applications that can process Big Data in parallel on multiple nodes. MapReduce provides analytical capabilities for analyzing huge volumes of complex data.

What is Big Data?

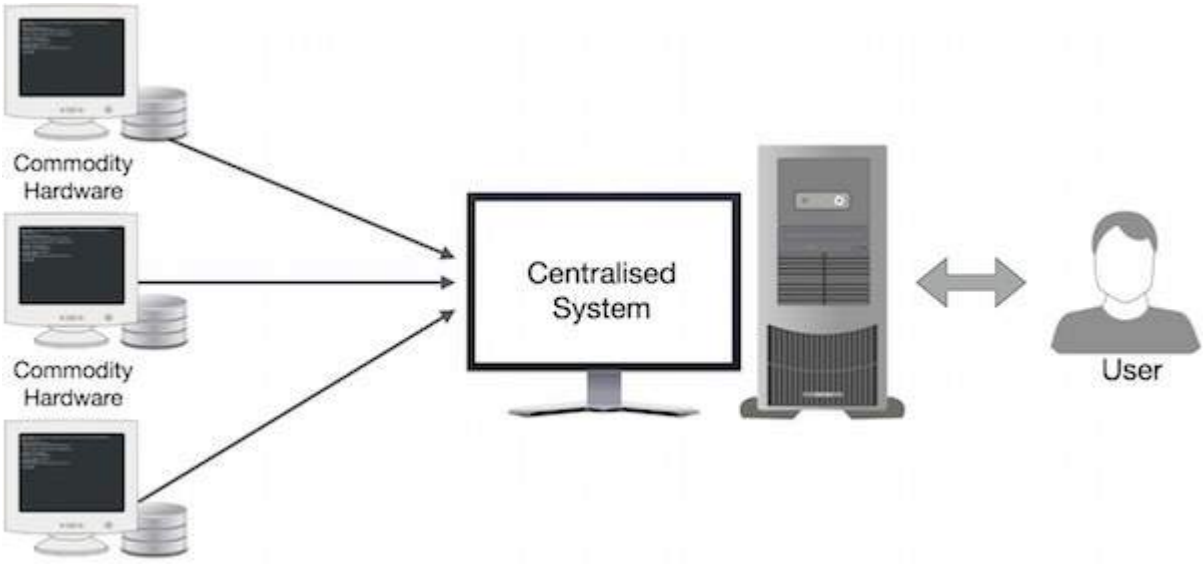
Big Data is a collection of large datasets that cannot be processed using traditional computing techniques. For example, the volume of data Facebook or Youtube need require it to collect and manage on a daily basis, can fall under the category of Big Data. However, Big Data is not only about scale and volume, it also involves one or more of the following aspects − Velocity, Variety, Volume, and Complexity.

Why MapReduce?

Traditional Enterprise Systems normally have a centralized server to store and process data. The following illustration depicts a schematic view of a traditional enterprise system. Traditional model is certainly not suitable to process huge volumes of scalable data and cannot be accommodated by standard database servers. Moreover, the centralized system creates too much of a bottleneck while processing multiple files simultaneously.



Google solved this bottleneck issue using an algorithm called MapReduce. MapReduce divides a task into small parts and assigns them to many computers. Later, the results are collected at one place and integrated to form the result dataset.



How MapReduce Works?

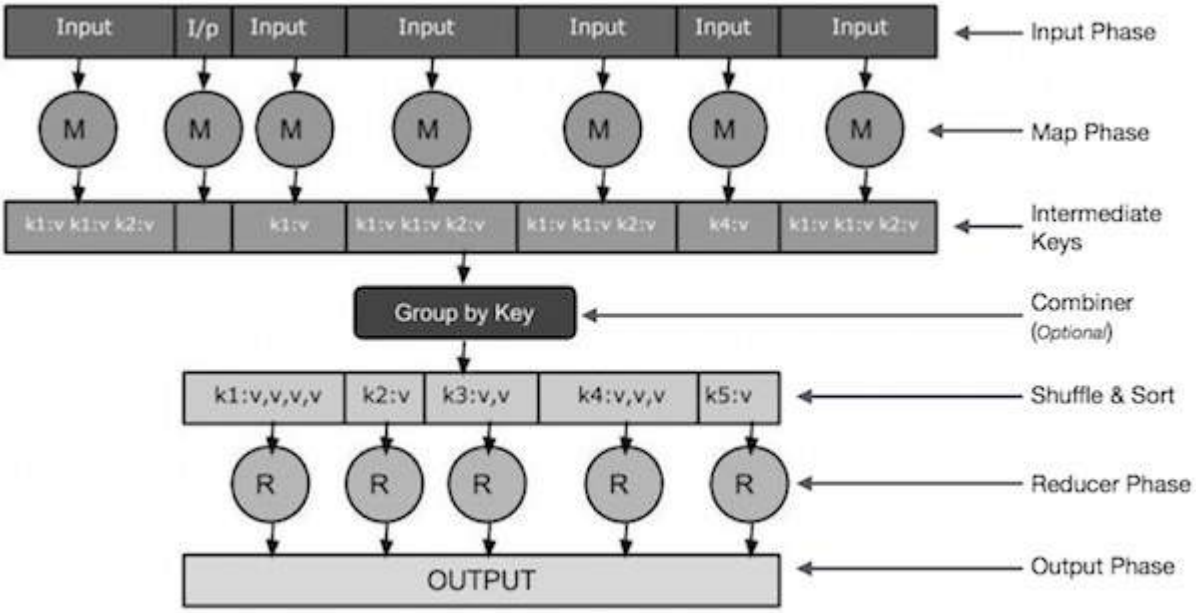
The MapReduce algorithm contains two important tasks, namely Map and Reduce.

- The Map task takes a set of data and converts it into another set of data, where individual elements are broken down into tuples *key − valuepairs*.

- The Reduce task takes the output from the Map as an input and combines those data tuples *key − valuepairs* into a smaller set of tuples.

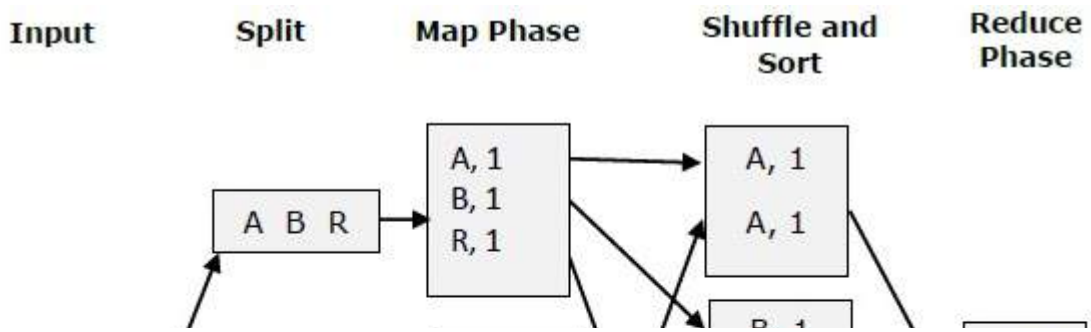
The reduce task is always performed after the map job.

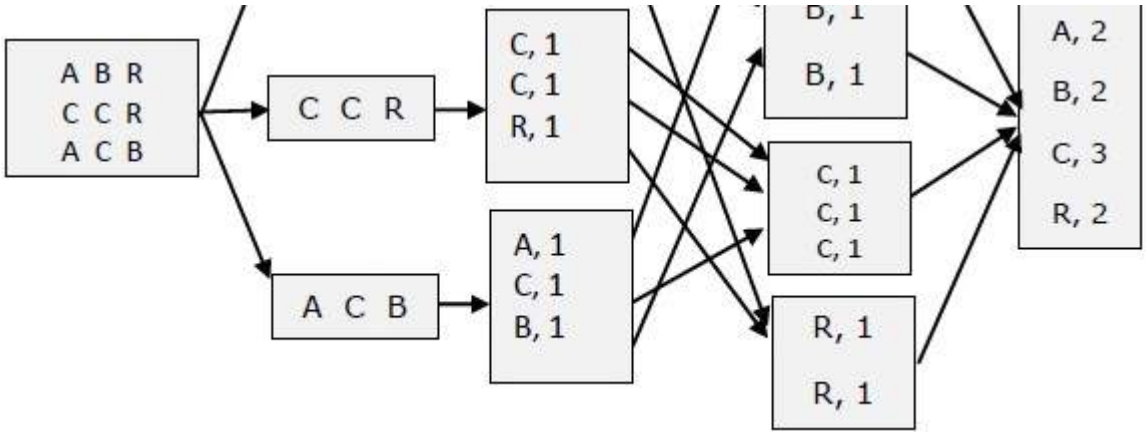
Let us now take a close look at each of the phases and try to understand their significance.



- **Input Phase** − Here we have a Record Reader that translates each record in an input file and sends the parsed data to the mapper in the form of key-value pairs.
- **Map** − Map is a user-defined function, which takes a series of key-value pairs and processes each one of them to generate zero or more key-value pairs.
- **Intermediate Keys** − They key-value pairs generated by the mapper are known as intermediate keys.
- **Combiner** − A combiner is a type of local Reducer that groups similar data from the map phase into identifiable sets. It takes the intermediate keys from the mapper as input and applies a user-defined code to aggregate the values in a small scope of one mapper. It is not a part of the main MapReduce algorithm; it is optional.
- **Shuffle and Sort** − The Reducer task starts with the Shuffle and Sort step. It downloads the grouped key-value pairs onto the local machine, where the Reducer is running. The individual key-value pairs are sorted by key into a larger data list. The data list groups the equivalent keys together so that their values can be iterated easily in the Reducer task.
- **Reducer** − The Reducer takes the grouped key-value paired data as input and runs a Reducer function on each one of them. Here, the data can be aggregated, filtered, and combined in a number of ways, and it requires a wide range of processing. Once the execution is over, it gives zero or more key-value pairs to the final step.
- **Output Phase** − In the output phase, we have an output formatter that translates the final key-value pairs from the Reducer function and writes them onto a file using a record writer.

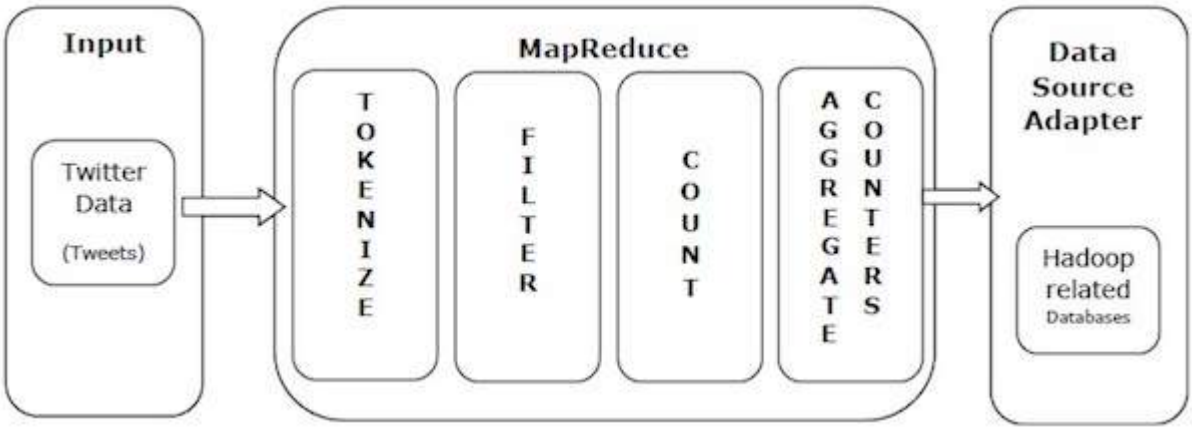
Let us try to understand the two tasks Map &f Reduce with the help of a small diagram −





MapReduce-Example

Let us take a real-world example to comprehend the power of MapReduce. Twitter receives around 500 million tweets per day, which is nearly 3000 tweets per second. The following illustration shows how Tweeter manages its tweets with the help of MapReduce.



As shown in the illustration, the MapReduce algorithm performs the following actions −

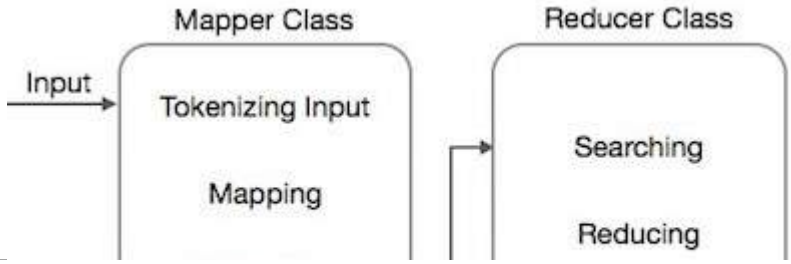
- **Tokenize** − Tokenizes the tweets into maps of tokens and writes them as key-value pairs.
- **Filter** − Filters unwanted words from the maps of tokens and writes the filtered maps as key-value pairs.
- **Count** − Generates a token counter per word.
- **Aggregate Counters** − Prepares an aggregate of similar counter values into small manageable units.

MAPREDUCE - ALGORITHM

The MapReduce algorithm contains two important tasks, namely Map and Reduce.

- The map task is done by means of Mapper Class
- The reduce task is done by means of Reducer Class.

Mapper class takes the input, tokenizes it, maps and sorts it. The output of Mapper class is used as input by Reducer class, which in turn searches matching pairs and reduces them.



MapReduce implements various mathematical algorithms to divide a task into small parts and assign them to multiple systems. In technical terms, MapReduce algorithm helps in sending the Map & Reduce tasks to appropriate servers in a cluster.

These mathematical algorithms may include the following −

- Sorting
- Searching
- Indexing
- TF-IDF

Sorting

Sorting is one of the basic MapReduce algorithms to process and analyze data. MapReduce implements sorting algorithm to automatically sort the output key-value pairs from the mapper by their keys.

- Sorting methods are implemented in the mapper class itself.
- In the Shuffle and Sort phase, after tokenizing the values in the mapper class, the **Context** class $user-definedclass$ collects the matching valued keys as a collection.
- To collect similar key-value pairs $intermediatekeys$, the Mapper class takes the help of **RawComparator** class to sort the key-value pairs.
- The set of intermediate key-value pairs for a given Reducer is automatically sorted by Hadoop to form key-values $K2, V2, V2, \ldots$ before they are presented to the Reducer.

Searching

Searching plays an important role in MapReduce algorithm. It helps in the combiner phase $optional$ and in the Reducer phase. Let us try to understand how Searching works with the help of an example.

Example

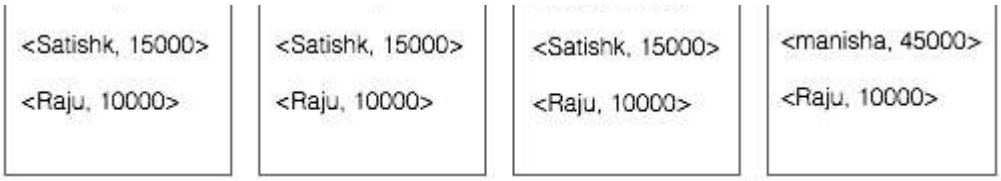
The following example shows how MapReduce employs Searching algorithm to find out the details of the employee who draws the highest salary in a given employee dataset.

- Let us assume we have employee data in four different files − A, B, C, and D. Let us also assume there are duplicate employee records in all four files because of importing the employee data from all database tables repeatedly. See the following illustration.

name, salary	name, salary	name, salary	name, salary
satish, 26000	gopal, 50000	satish, 26000	satish, 26000
Krishna, 25000	Krishna, 25000	kiran, 45000	Krishna, 25000
Satishk, 15000	Satishk, 15000	Satishk, 15000	manisha, 45000
Raju, 10000	Raju, 10000	Raju, 10000	Raju, 10000

- **The Map phase** processes each input file and provides the employee data in key-value pairs $< k, v >:< empname, salary >$. See the following illustration.

<satish, 26000>	<gopal, 50000>	<satish, 26000>	<satish, 26000>
<Krishna, 25000>	<Krishna, 25000>	<kiran, 45000>	<Krishna, 25000>



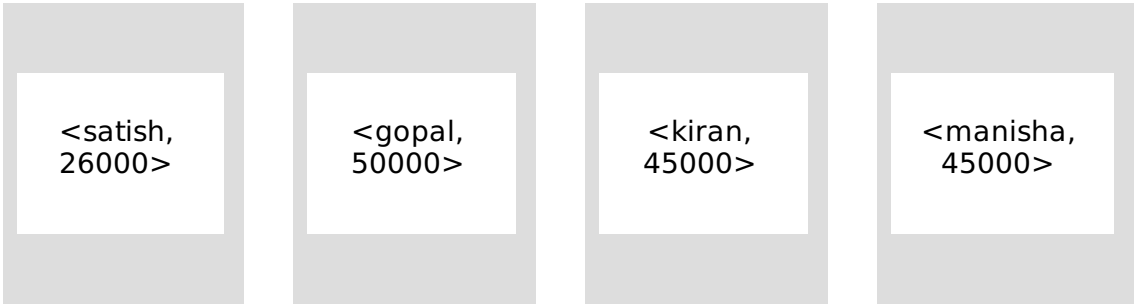
- **The combiner phase** *searchingtechnique* will accept the input from the Map phase as a key-value pair with employee name and salary. Using searching technique, the combiner will check all the employee salary to find the highest salaried employee in each file. See the following snippet.

```
<k: employee name, v: salary>
Max= the salary of an first employee. Treated as max salary

if(v(second employee).salary > Max){
   Max = v(salary);
}

else{
   Continue checking;
}
```

The expected result is as follows −



- **Reducer phase** − Form each file, you will find the highest salaried employee. To avoid redundancy, check all the <k, v> pairs and eliminate duplicate entries, if any. The same algorithm is used in between the four <k, v> pairs, which are coming from four input files. The final output should be as follows −

```
<gopal, 50000>
```

Indexing

Normally indexing is used to point to a particular data and its address. It performs batch indexing on the input files for a particular Mapper.

The indexing technique that is normally used in MapReduce is known as **inverted index.** Search engines like Google and Bing use inverted indexing technique. Let us try to understand how Indexing works with the help of a simple example.

Example

The following text is the input for inverted indexing. Here T[0], T[1], and t[2] are the file names and their content are in double quotes.

```
T[0] = "it is what it is"
T[1] = "what is it"
T[2] = "it is a banana"
```

After applying the Indexing algorithm, we get the following output −

```
"a": {2}
"banana": {2}
"is": {0, 1, 2}
"it": {0, 1, 2}
"what": {0, 1}
```

Here "a": {2} implies the term "a" appears in the T[2] file. Similarly, "is": {0, 1, 2} implies the term "is" appears in the files T[0], T[1], and T[2].

TF-IDF

TF-IDF is a text processing algorithm which is short for Term Frequency − Inverse Document Frequency. It is one of the common web analysis algorithms. Here, the term 'frequency' refers to the number of times a term appears in a document.

Term Frequency *TF*

It measures how frequently a particular term occurs in a document. It is calculated by the number of times a word appears in a document divided by the total number of words in that document.

```
TF(the) = (Number of times term the ‘the’ appears in a document) / (Total number of terms
in the document)
```

Inverse Document Frequency *IDF*

It measures the importance of a term. It is calculated by the number of documents in the text database divided by the number of documents where a specific term appears.

While computing TF, all the terms are considered equally important. That means, TF counts the term frequency for normal words like “is”, “a”, “what”, etc. Thus we need to know the frequent terms while scaling up the rare ones, by computing the following −

```
IDF(the) = log_e(Total number of documents / Number of documents with term ‘the’ in it).
```

The algorithm is explained below with the help of a small example.

Example

Consider a document containing 1000 words, wherein the word **hive** appears 50 times. The TF for **hive** is then $50/1000 = 0.05$.

Now, assume we have 10 million documents and the word **hive** appears in 1000 of these. Then, the IDF is calculated as $\log 10,000,000/1,000 = 4$.

The TF-IDF weight is the product of these quantities − $0.05 \times 4 = 0.20$.

MAPREDUCE - INSTALLATION

MapReduce works only on Linux flavored operating systems and it comes inbuilt with a Hadoop Framework. We need to perform the following steps in order to install Hadoop framework.

Verifying JAVA Installation

Java must be installed on your system before installing Hadoop. Use the following command to check whether you have Java installed on your system.

```
$ java –version
```

If Java is already installed on your system, you get to see the following response −

```
java version "1.7.0_71"
Java(TM) SE Runtime Environment (build 1.7.0_71-b13)
Java HotSpot(TM) Client VM (build 25.0-b02, mixed mode)
```

In case you don't have Java installed on your system, then follow the steps given below.

Installing Java

Step 1

Download the latest version of Java from the following link − this link.

After downloading, you can locate the file **jdk-7u71-linux-x64.tar.gz** in your Downloads folder.

Step 2

Use the following commands to extract the contents of jdk-7u71-linux-x64.gz.

```
$ cd Downloads/
$ ls
jdk-7u71-linux-x64.gz
$ tar zxf jdk-7u71-linux-x64.gz
$ ls
jdk1.7.0_71 jdk-7u71-linux-x64.gz
```

Step 3

To make Java available to all the users, you have to move it to the location "/usr/local/". Go to root and type the following commands −

```
$ su
password:
# mv jdk1.7.0_71 /usr/local/java
# exit
```

Step 4

For setting up PATH and JAVA_HOME variables, add the following commands to ~/.bashrc file.

```
export JAVA_HOME=/usr/local/java
export PATH=$PATH:$JAVA_HOME/bin
```

Apply all the changes to the current running system.

```
$ source ~/.bashrc
```

Step 5

Use the following commands to configure Java alternatives −

```
# alternatives --install /usr/bin/java java usr/local/java/bin/java 2

# alternatives --install /usr/bin/javac javac usr/local/java/bin/javac 2

# alternatives --install /usr/bin/jar jar usr/local/java/bin/jar 2

# alternatives --set java usr/local/java/bin/java

# alternatives --set javac usr/local/java/bin/javac

# alternatives --set jar usr/local/java/bin/jar
```

Now verify the installation using the command **java -version** from the terminal.

Verifying Hadoop Installation

Hadoop must be installed on your system before installing MapReduce. Let us verify the Hadoop installation using the following command −

```
$ hadoop version
```

If Hadoop is already installed on your system, then you will get the following response −

```
Hadoop 2.4.1
--
Subversion https://svn.apache.org/repos/asf/hadoop/common -r 1529768
Compiled by hortonmu on 2013-10-07T06:28Z
Compiled with protoc 2.5.0
From source with checksum 79e53ce7994d1628b240f09af91e1af4
```

If Hadoop is not installed on your system, then proceed with the following steps.

Downloading Hadoop

Download Hadoop 2.4.1 from Apache Software Foundation and extract its contents using the following commands.

```
$ su
password:
# cd /usr/local
# wget http://apache.claz.org/hadoop/common/hadoop-2.4.1/
hadoop-2.4.1.tar.gz
# tar xzf hadoop-2.4.1.tar.gz
# mv hadoop-2.4.1/* to hadoop/
# exit
```

Installing Hadoop in Pseudo Distributed mode

The following steps are used to install Hadoop 2.4.1 in pseudo distributed mode.

Step 1 − Setting up Hadoop

You can set Hadoop environment variables by appending the following commands to ~/.bashrc file.

```
export HADOOP_HOME=/usr/local/hadoop
export HADOOP_MAPRED_HOME=$HADOOP_HOME
export HADOOP_COMMON_HOME=$HADOOP_HOME
export HADOOP_HDFS_HOME=$HADOOP_HOME
export YARN_HOME=$HADOOP_HOME
export HADOOP_COMMON_LIB_NATIVE_DIR=$HADOOP_HOME/lib/native
export PATH=$PATH:$HADOOP_HOME/sbin:$HADOOP_HOME/bin
```

Apply all the changes to the current running system.

```
$ source ~/.bashrc
```

Step 2 − Hadoop Configuration

You can find all the Hadoop configuration files in the location “$HADOOP_HOME/etc/hadoop”. You need to make suitable changes in those configuration files according to your Hadoop infrastructure.

```
$ cd $HADOOP_HOME/etc/hadoop
```

In order to develop Hadoop programs using Java, you have to reset the Java environment variables in **hadoop-env.sh** file by replacing JAVA_HOME value with the location of Java in your system.

```
export JAVA_HOME=/usr/local/java
```

You have to edit the following files to configure Hadoop −

- core-site.xml
- hdfs-site.xml
- yarn-site.xml
- mapred-site.xml

core-site.xml

core-site.xml contains the following information−

- Port number used for Hadoop instance
- Memory allocated for the file system
- Memory limit for storing the data
- Size of Read/Write buffers

Open the core-site.xml and add the following properties in between the <configuration> and </configuration> tags.

```
<configuration>
   <property>
      <name>fs.default.name</name>
      <value>hdfs://localhost:9000 </value>
   </property>
</configuration>
```

hdfs-site.xml

hdfs-site.xml contains the following information −

- Value of replication data
- The namenode path
- The datanode path of your local file systems *theplacewhereyouwanttostoretheHadoopinfra*

Let us assume the following data.

```
dfs.replication (data replication value) = 1

(In the following path /hadoop/ is the user name.
hadoopinfra/hdfs/namenode is the directory created by hdfs file system.)
namenode path = //home/hadoop/hadoopinfra/hdfs/namenode

(hadoopinfra/hdfs/datanode is the directory created by hdfs file system.)
datanode path = //home/hadoop/hadoopinfra/hdfs/datanode
```

Open this file and add the following properties in between the <configuration>, </configuration> tags.

```
<configuration>

   <property>
      <name>dfs.replication</name>
      <value>1</value>
   </property>
```

```
   <property>
      <name>dfs.name.dir</name>
      <value>file:///home/hadoop/hadoopinfra/hdfs/namenode</value>
   </property>

   <property>
      <name>dfs.data.dir</name>
      <value>file:///home/hadoop/hadoopinfra/hdfs/datanode </value>
   </property>

</configuration>
```

Note − In the above file, all the property values are user-defined and you can make changes according to your Hadoop infrastructure.

yarn-site.xml

This file is used to configure yarn into Hadoop. Open the yarn-site.xml file and add the following properties in between the <configuration>, </configuration> tags.

```
<configuration>
   <property>
      <name>yarn.nodemanager.aux-services</name>
      <value>mapreduce_shuffle</value>
   </property>
</configuration>
```

mapred-site.xml

This file is used to specify the MapReduce framework we are using. By default, Hadoop contains a template of yarn-site.xml. First of all, you need to copy the file from mapred-site.xml.template to mapred-site.xml file using the following command.

```
$ cp mapred-site.xml.template mapred-site.xml
```

Open mapred-site.xml file and add the following properties in between the <configuration>, </configuration> tags.

```
<configuration>
   <property>
      <name>mapreduce.framework.name</name>
      <value>yarn</value>
   </property>
</configuration>
```

Verifying Hadoop Installation

The following steps are used to verify the Hadoop installation.

Step 1 − Name Node Setup

Set up the namenode using the command “hdfs namenode -format” as follows −

```
$ cd ~
$ hdfs namenode -format
```

The expected result is as follows −

```
10/24/14 21:30:55 INFO namenode.NameNode: STARTUP_MSG:
/************************************************************
STARTUP_MSG: Starting NameNode
STARTUP_MSG: host = localhost/192.168.1.11
STARTUP_MSG: args = [-format]
STARTUP_MSG: version = 2.4.1
```

```
...
...
10/24/14 21:30:56 INFO common.Storage: Storage directory
/home/hadoop/hadoopinfra/hdfs/namenode has been successfully formatted.
10/24/14 21:30:56 INFO namenode.NNStorageRetentionManager: Going to
retain 1 images with txid >= 0
10/24/14 21:30:56 INFO util.ExitUtil: Exiting with status 0
10/24/14 21:30:56 INFO namenode.NameNode: SHUTDOWN_MSG:

/************************************************************
SHUTDOWN_MSG: Shutting down NameNode at localhost/192.168.1.11
************************************************************/
```

Step 2 − Verifying Hadoop dfs

Execute the following command to start your Hadoop file system.

```
$ start-dfs.sh
```

The expected output is as follows −

```
10/24/14 21:37:56
Starting namenodes on [localhost]
localhost: starting namenode, logging to /home/hadoop/hadoop-
2.4.1/logs/hadoop-hadoop-namenode-localhost.out
localhost: starting datanode, logging to /home/hadoop/hadoop-
2.4.1/logs/hadoop-hadoop-datanode-localhost.out
Starting secondary namenodes [0.0.0.0]
```

Step 3 − Verifying Yarn Script

The following command is used to start the yarn script. Executing this command will start your yarn daemons.

```
$ start-yarn.sh
```

The expected output is as follows −

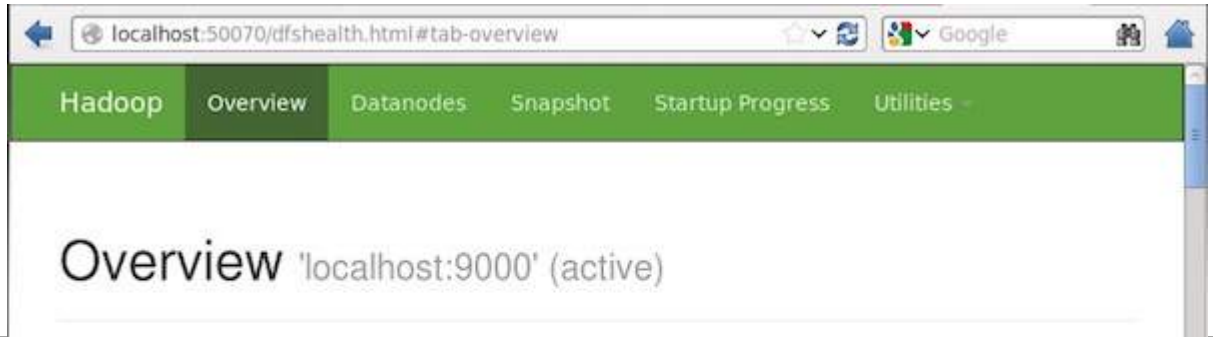
```
starting yarn daemons
starting resourcemanager, logging to /home/hadoop/hadoop-
2.4.1/logs/yarn-hadoop-resourcemanager-localhost.out
localhost: starting node manager, logging to /home/hadoop/hadoop-
2.4.1/logs/yarn-hadoop-nodemanager-localhost.out
```

Step 4 − Accessing Hadoop on Browser

The default port number to access Hadoop is 50070. Use the following URL to get Hadoop services on your browser.

```
http://localhost:50070/
```

The following screenshot shows the Hadoop browser.



Step 5 − Verify all Applications of a Cluster

The default port number to access all the applications of a cluster is 8088. Use the following URL to use this service.

```
http://localhost:8088/
```

The following screenshot shows a Hadoop cluster browser.

MAPREDUCE - API

In this chapter, we will take a close look at the classes and their methods that are involved in the operations of MapReduce programming. We will primarily keep our focus on the following −

- JobContext Interface
- Job Class
- Mapper Class
- Reducer Class

JobContext Interface

The JobContext interface is the super interface for all the classes, which defines different jobs in MapReduce. It gives you a read-only view of the job that is provided to the tasks while they are running.

The following are the sub-interfaces of JobContext interface.

S.No.	Subinterface Description
1.	**MapContext<KEYIN, VALUEIN, KEYOUT, VALUEOUT>** Defines the context that is given to the Mapper.
2.	**ReduceContext<KEYIN, VALUEIN, KEYOUT, VALUEOUT>** Defines the context that is passed to the Reducer.

Job class is the main class that implements the JobContext interface.

Job Class

The Job class is the most important class in the MapReduce API. It allows the user to configure the job, submit it, control its execution, and query the state. The set methods only work until the job is submitted, afterwards they will throw an IllegalStateException.

Normally, the user creates the application, describes the various facets of the job, and then submits the job and monitors its progress.

Here is an example of how to submit a job −

```
// Create a new Job
Job job = new Job(new Configuration());
job.setJarByClass(MyJob.class);

// Specify various job-specific parameters
job.setJobName("myjob");
job.setInputPath(new Path("in"));
job.setOutputPath(new Path("out"));

job.setMapperClass(MyJob.MyMapper.class);
job.setReducerClass(MyJob.MyReducer.class);

// Submit the job, then poll for progress until the job is complete
job.waitForCompletion(true);
```

Constructors

Following are the constructor summary of Job class.

S.No	Constructor Summary
1	**Job**
2	**Job***Configurationconf*
3	**Job***Configurationconf, StringjobName*

Methods

Some of the important methods of Job class are as follows −

S.No	Method Description
1	**getJobName** User-specified job name.
2	**getJobState** Returns the current state of the Job.
3	**isComplete** Checks if the job is finished or not.
4	**setInputFormatClass** Sets the InputFormat for the job.

5	**setJobName***Stringname* Sets the user-specified job name.
6	**setOutputFormatClass** Sets the Output Format for the job.
7	**setMapperClass***Class* Sets the Mapper for the job.
8	**setReducerClass***Class* Sets the Reducer for the job.
9	**setPartitionerClass***Class* Sets the Partitioner for the job.
10	**setCombinerClass***Class* Sets the Combiner for the job.

Mapper Class

The Mapper class defines the Map job. Maps input key-value pairs to a set of intermediate key-value pairs. Maps are the individual tasks that transform the input records into intermediate records. The transformed intermediate records need not be of the same type as the input records. A given input pair may map to zero or many output pairs.

Method

map is the most prominent method of the Mapper class. The syntax is defined below −

```
map(KEYIN key, VALUEIN value, org.apache.hadoop.mapreduce.Mapper.Context context)
```

This method is called once for each key-value pair in the input split.

Reducer Class

The Reducer class defines the Reduce job in MapReduce. It reduces a set of intermediate values that share a key to a smaller set of values. Reducer implementations can access the Configuration for a job via the JobContext.getConfiguration method. A Reducer has three primary phases − Shuffle, Sort, and Reduce.

- **Shuffle** − The Reducer copies the sorted output from each Mapper using HTTP across the network.
- **Sort** − The framework merge-sorts the Reducer inputs by keys *sincedifferentMappersmayhaveoutputthesamekey*. The shuffle and sort phases occur simultaneously, i.e., while outputs are being fetched, they are merged.
- **Reduce** − In this phase the reduce *Object, Iterable, Context* method is called for each <key, *collectionofvalues*> in the sorted inputs.

Method

reduce is the most prominent method of the Reducer class. The syntax is defined below −

```
reduce(KEYIN key, Iterable<VALUEIN> values, org.apache.hadoop.mapreduce.Reducer.Context
context)
```

This method is called once for each key on the collection of key-value pairs.

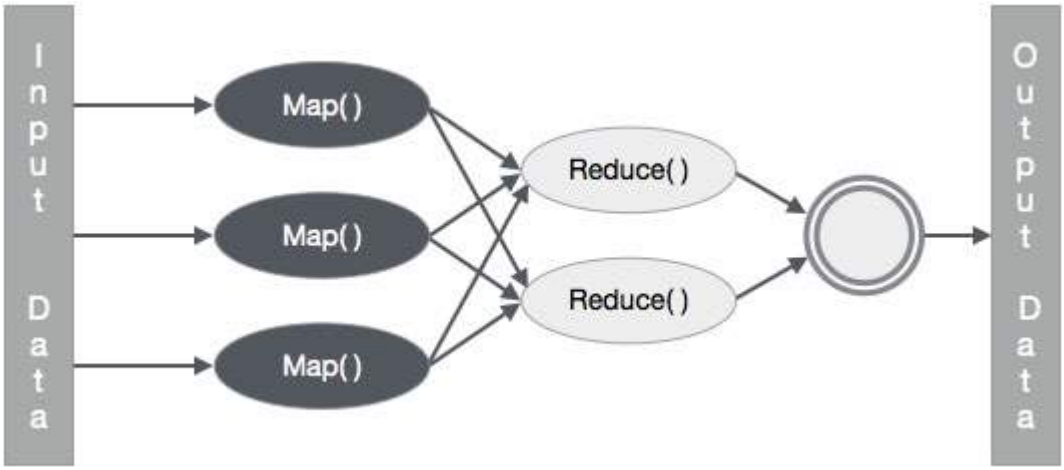
MAPREDUCE - HADOOP IMPLEMENTATION

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
```

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*, *below*20, *between*21*to*30, *above*30.

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 + filename + line number" example: key = @input1 and the value would be the data in that line example: value = 1201 \t gopal \t 45 \t Male \t 50000.

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 for example, /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
```

MAPREDUCE - COMBINERS

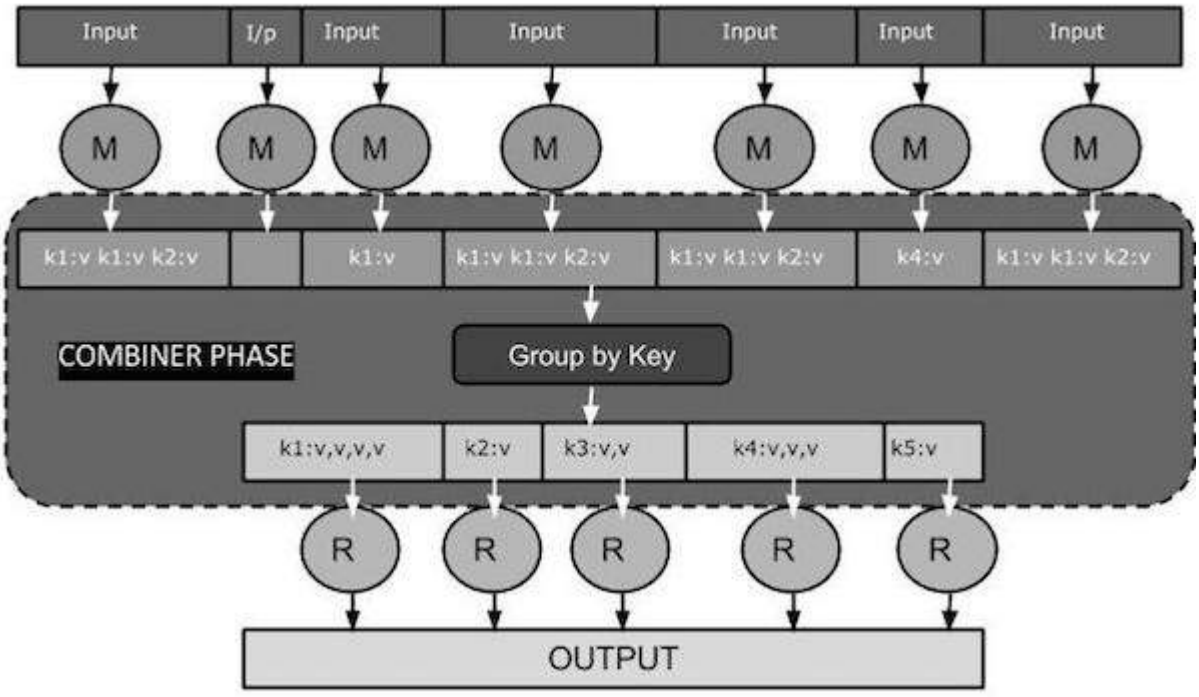
A Combiner, also known as a **semi-reducer,** is an optional class that operates by accepting the inputs from the Map class and thereafter passing the output key-value pairs to the Reducer class.

The main function of a Combiner is to summarize the map output records with the same key. The output key-value collection 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 for example, /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
```

MAPREDUCE - HADOOP ADMINISTRATION

This chapter explains Hadoop administration which includes both HDFS and MapReduce administration.

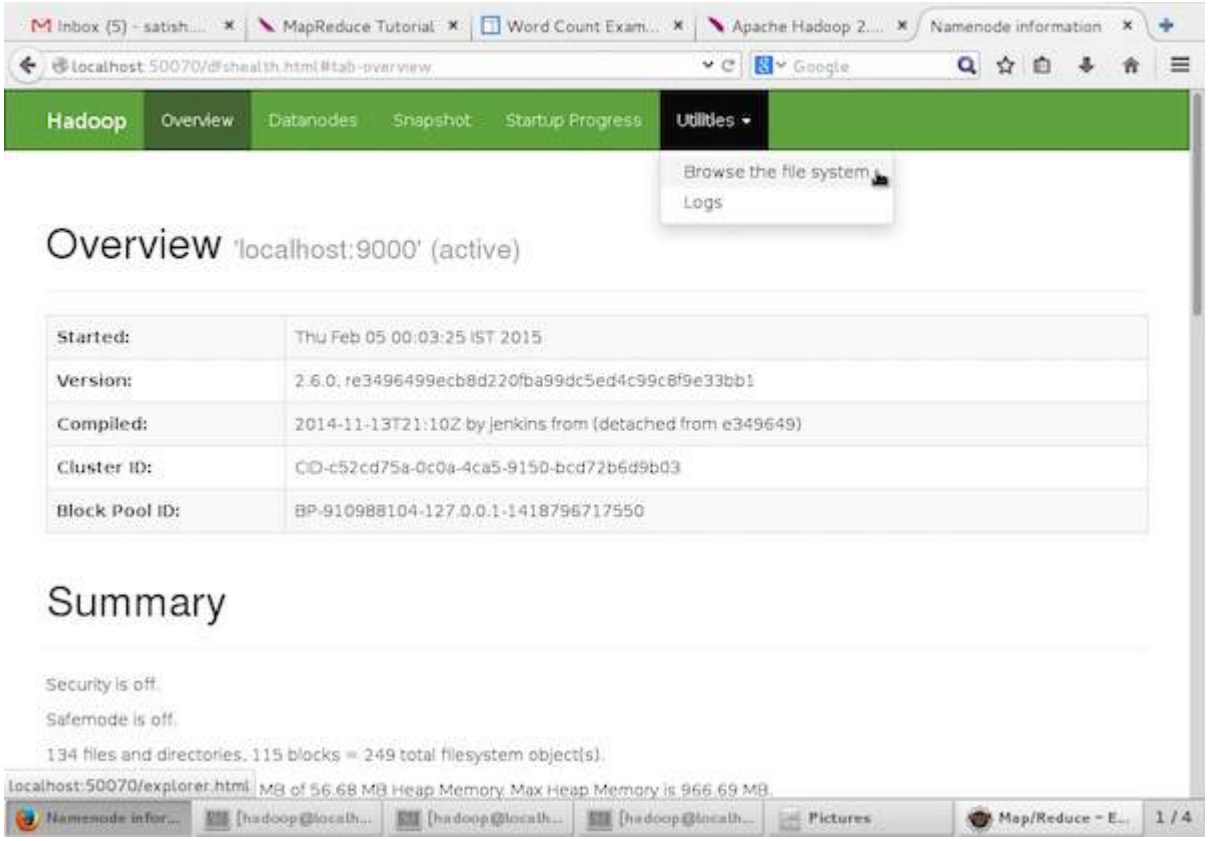
- HDFS administration includes monitoring the HDFS file structure, locations, and the updated files.
- MapReduce administration includes monitoring the list of applications, configuration of nodes, application status, etc.

HDFS Monitoring

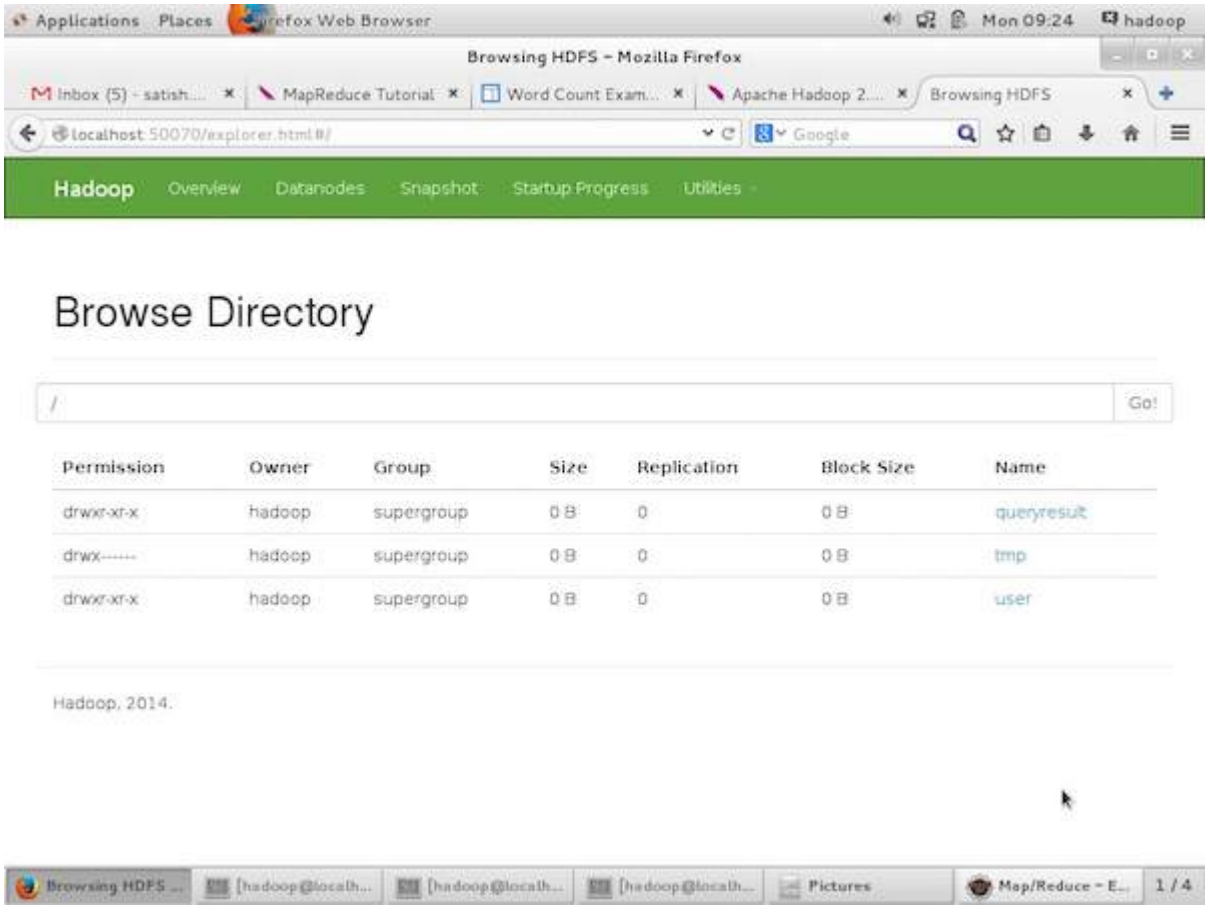
HDFS Hadoop Distributed File System contains the user directories, input files, and output files. Use the MapReduce commands, **put** and **get,** for storing and retrieving.

After starting the Hadoop framework daemons by passing the command “start-all.sh” on “/$HADOOP_HOME/sbin”, pass the following URL to the browser “http://localhost:50070”. You should see the following screen on your browser.

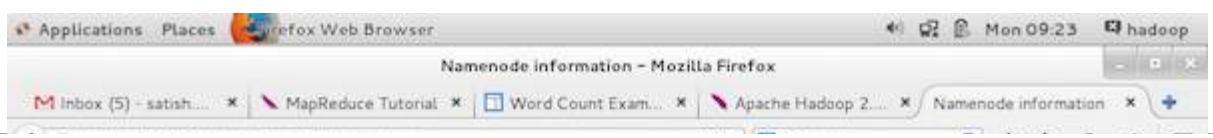
The following screenshot shows how to browse the browse HDFS.

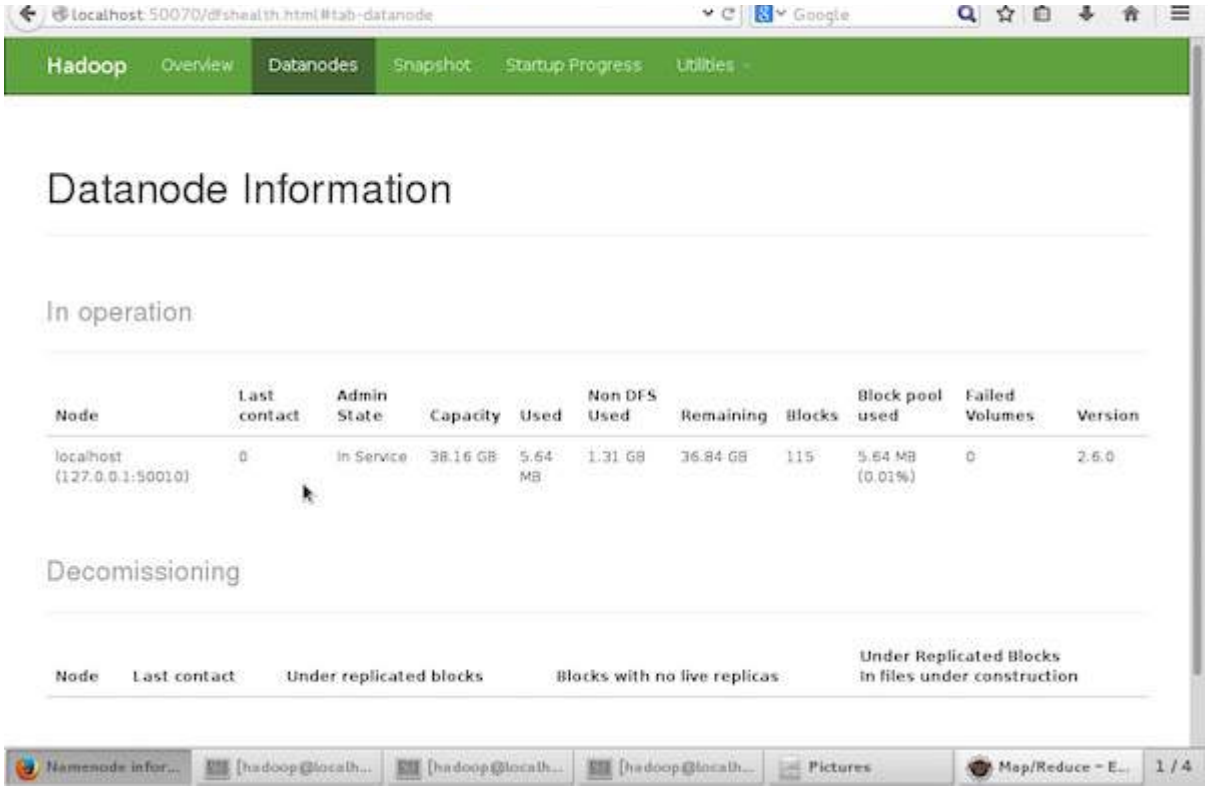


The following screenshot show the file structure of HDFS. It shows the files in the "/user/hadoop" directory.



The following screenshot shows the Datanode information in a cluster. Here you can find one node with its configurations and capacities.



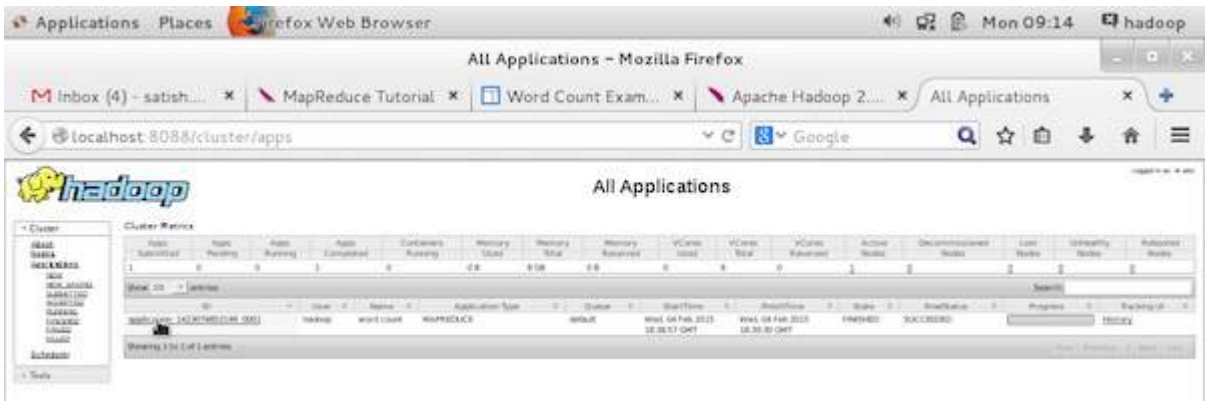


MapReduce Job Monitoring

A MapReduce application is a collection of jobs Map job, Combiner, Partitioner, and Reduce job. It is mandatory to monitor and maintain the following −

- Configuration of datanode where the application is suitable.
- The number of datanodes and resources used per application.

To monitor all these things, it is imperative that we should have a user interface. After starting the Hadoop framework by passing the command “start-all.sh” on “/$HADOOP_HOME/sbin”, pass the following URL to the browser “http://localhost:8080”. You should see the following screen on your browser.

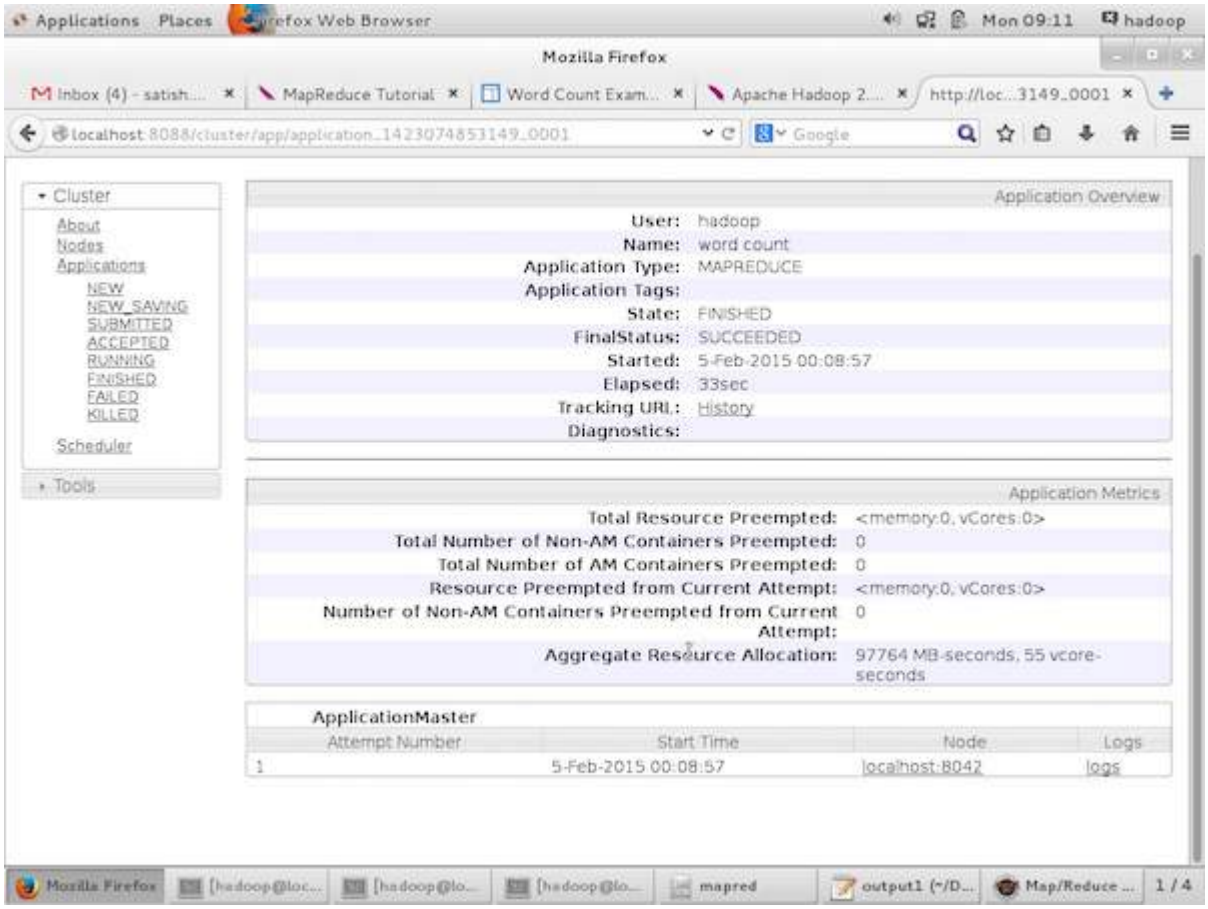


In the above screenshot, the hand pointer is on the application ID. Just click on it to find the following screen on your browser. It describes the following −

- On which user the current application is running
- The application name
- Type of that application
- Current status, Final status
- Application started time, elapsed completed time, if it is complete at the time of monitoring
- The history of this application, i.e., log information

- And finally, the node information, i.e., the nodes that participated in running the application.

The following screenshot shows the details of a particular application −



The following screenshot describes the currently running nodes information. Here, the screenshot contains only one node. A hand pointer shows the localhost address of the running node.