Apache Hadoop

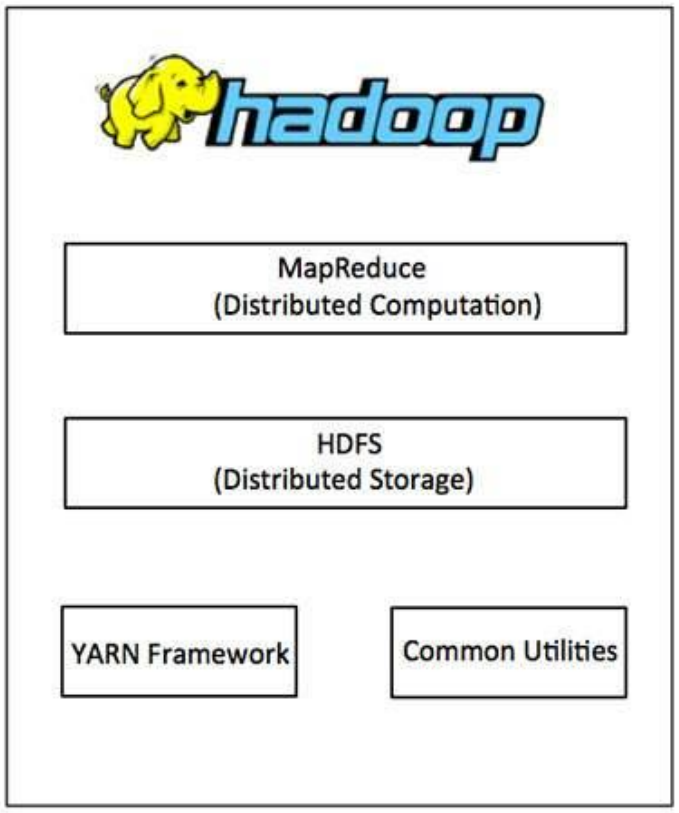
Hadoop is an Apache open source framework written in java that allows distributed processing of large datasets across clusters of computers using simple programming models. A Hadoop frame-worked application works in an environment that provides distributed storage and computation across clusters of computers. Hadoop is designed to scale up from single server to thousands of machines, each offering local computation and storage.

Hadoop Architecture

Hadoop framework includes following four modules:

- **Hadoop Common:** These are Java libraries and utilities required by other Hadoop modules. These libraries provides file system and OS level abstractions and contains the necessary Java files and scripts required to start Hadoop.
- **Hadoop YARN:** This is a framework for job scheduling and cluster resource management.
- **Hadoop Distributed File System** HDFS™HDFS™**:** A distributed file system that provides high-throughput access to application data.
- **Hadoop MapReduce:** This is YARN-based system for parallel processing of large data sets.

We can use following diagram to depict these four components available in Hadoop framework.



Since 2012, the term "Hadoop" often refers not just to the base modules mentioned above but also to the collection of additional software packages that can be installed on top of or alongside Hadoop, such as Apache Pig, Apache Hive, Apache HBase, Apache Spark etc.

MapReduce

Hadoop **MapReduce** is a software framework for easily writing applications which process big amounts of data in-parallel on large clusters thousands of nodes thousands of nodes of commodity hardware in a reliable, fault-tolerant manner.

The term MapReduce actually refers to the following two different tasks that Hadoop programs perform:

- **The Map Task:** This is the first task, which takes input data and converts it into a set of data, where individual elements are broken down into tuples key/value pairs.
- **The Reduce Task:** This task takes the output from a map task as input and combines those data tuples into a smaller set of tuples. The reduce task is always performed after the map task.

Typically both the input and the output are stored in a file-system. The framework takes care of scheduling tasks, monitoring them and re-executes the failed tasks.

The MapReduce framework consists of a single master **JobTracker** and one slave **TaskTracker** per cluster-node. The master is responsible for resource management, tracking resource consumption/availability and scheduling the jobs component tasks on the slaves, monitoring them and re-executing the failed tasks. The slaves TaskTracker execute the tasks as directed by the master and provide task-status information to the master periodically.

The JobTracker is a single point of failure for the Hadoop MapReduce service which means if JobTracker goes down, all running jobs are halted.

Hadoop Distributed File System

Hadoop can work directly with any mountable distributed file system such as Local FS, HFTP FS, S3 FS, and others, but the most common file system used by Hadoop is the Hadoop Distributed File System HDFSHDFS.

The Hadoop Distributed File System HDFSHDFS is based on the Google File System GFSGFS and provides a distributed file system that is designed to run on large clusters thousands of computers of small computer machines in a reliable, fault-tolerant manner.

HDFS uses a master/slave architecture where master consists of a single **NameNode** that manages the file system metadata and one or more slave **DataNodes** that store the actual data.

A file in an HDFS namespace is split into several blocks and those blocks are stored in a set of DataNodes. The NameNode determines the mapping of blocks to the DataNodes. The DataNodes takes care of read and write operation with the file system. They also take care of block creation, deletion and replication based on instruction given by NameNode.

HDFS provides a shell like any other file system and a list of commands are available to interact with the file system. These shell commands will be covered in a separate chapter along with appropriate examples.

How Does Hadoop Work?

Stage 1

A user/application can submit a job to the Hadoop job client for required process by specifying the following items:

1. The location of the input and output files in the distributed file system.
2. The java classes in the form of jar file containing the implementation of map and reduce functions.
3. The job configuration by setting different parameters specific to the job.

Stage 2

The Hadoop job client then submits the job jar/executable etc and configuration to the JobTracker which then assumes the responsibility of distributing the software/configuration to the slaves, scheduling tasks and monitoring them, providing status and diagnostic information to the job-client.

Stage 3

The TaskTrackers on different nodes execute the task as per MapReduce implementation and output of the reduce function is stored into the output files on the file system.

Advantages of Hadoop

- Hadoop framework allows the user to quickly write and test distributed systems. It is efficient, and it automatic distributes the data and work across the machines and in turn, utilizes the underlying parallelism of the CPU cores.
- Hadoop does not rely on hardware to provide fault-tolerance and high availability FTHAFTHA, rather Hadoop library itself has been designed to detect and handle failures at the application layer.
- Servers can be added or removed from the cluster dynamically and Hadoop continues to operate without interruption.
- Another big advantage of Hadoop is that apart from being open source, it is compatible on all the platforms since it is Java based.

Hadoop Ecosystem

Hadoop has gained its popularity due to its ability of storing, analysing and accessing large amount of data, quickly and cost effectively through clusters of commodity hardware. It won't be wrong if we say that Apache Hadoop is actually a collection of several components and not just a single product.

With Hadoop Ecosystem there are several commercial along with an open source products which are broadly used to make Hadoop laymen accessible and more usable. The following sections provide additional information on the individual components [28]:

Hive:

Hive is part of the Hadoop ecosystem and provides an SQL like interface to Hadoop. It is a data warehouse system for Hadoop that facilitates easy data summarization, ad-hoc queries, and the analysis of large datasets stored in Hadoop compatible file systems. It provides a mechanism to project structure onto this data and query the data using a SQL like language called HiveQL. Hive also allows traditional map/reduce programmers to plug in their custom map-pers and reducers when it is inconvenient or inefficient to express this logic in HiveQL.

Table 1. Components of Hadoop ecosystem

Component	Features/Description/Strength
Hive	Data warehouse with SQL-like access

HBase	Column-oriented database scaling to billions of rows
Zookeeper	Configuration management and coordination
Mahout	Library of machine learning and data mining algorithms
Sqoop	Imports data from relational databases
Spark	Fast and general engine for large-scale data processing
Pig	High-level programming language for Hadoop computations
Oozie	Orchestration and workflow management
Flume	Collection and import of log and event data
Ambari	Deployment, configuration and monitoring

HBase:

HBase is a distributed, column oriented database and uses HDFS for the underlying storage. HDFS works on write once and read many times pattern, but this isn't a case always. We may require real time read/write random access for huge dataset; this is where HBase comes into the picture. HBase is built on top of HDFS and distributed on column-oriented database.

ZooKeeper:

ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization and providing group services which are very useful for a variety of distributed systems. HBase is not operational without ZooKeeper.

Mahout:

Mahout is a scalable machine learning library that implements various different approaches machine learning. At present Mahout contains four main groups of algorithms:

Recommendations, also known as collective filtering

Classifications, also known as categorization

Clustering

Frequent item set mining, also known as parallel frequent pattern mining

Sqoop (SQL-to-Hadoop):

Sqoop is a tool designed for efficiently transferring structured data from SQL Server and SQL Azure to HDFS and then uses it in MapReduce and Hive jobs. One can even use Sqoop to move data from HDFS to SQL Server.

Apache Spark:

Apache Spark is a general compute engine that offers fast data analysis on a large scale. Spark is built on HDFS but bypasses MapReduce and instead uses its own data processing framework. Common uses cases for Apache Spark include real-time queries, event stream processing, iterative algorithms, complex operations and machine learning.

Pig:

Pig is a platform for analyzing and querying huge data sets that consist of a high-level language for expressing data analysis programs, coupled with infrastructure for evaluating these programs. Pig's built-in operations can make sense of semi-structured data, such as log files, and the language is extensible using Java to add support for custom data types and transformations.

Oozie:

Apache Oozie is a workflow/coordination system to manage Hadoop jobs.

Flume:

Flume is a framework for harvesting, aggregating and moving huge amounts of log data or text files in and out of Hadoop. Agents are populated throughout ones IT infrastructure inside web servers, application servers and mobile devices. Flume itself has a query processing engine, so it's easy to transform each new batch of data before it is shuttled to the intended sink.

Ambari:

Ambari was created to help manage Hadoop. It offers support for many of the tools in the Hadoop ecosystem including Hive, HBase, Pig, Sqoop and Zookeeper. The tool features a management dashboard that keeps track of cluster health and can help diagnose performance issues.

Thus concluding this discussion on frameworks it can be mentioned that Hadoop is powerful because it is extensible and it is easy to integrate with any component. Its popularity is due in part to its ability to store, analyze and access large amounts of data, quickly and cost effectively across clusters of commodity hardware. It is not actually a single product but instead a collection of several components. When all these components are merged, it makes the Hadoop very user friendly.

Moving Data in and Out of Hadoop

Hadoop File System was developed using distributed file system design. It is run on commodity hardware. Unlike other distributed systems, HDFS is highly fault tolerant and designed using low-cost hardware.

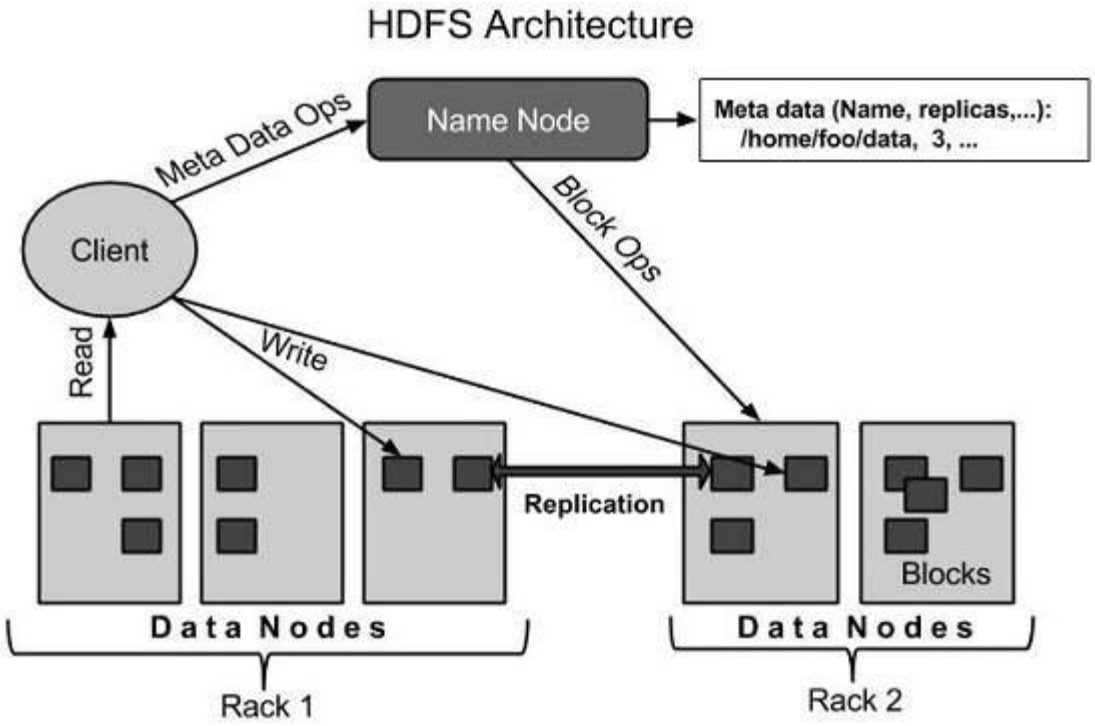
HDFS holds very large amount of data and provides easier access. To store such huge data, the files are stored across multiple machines. These files are stored in redundant fashion to rescue the system from possible data losses in case of failure. HDFS also makes applications available to parallel processing.

Features of HDFS

- It is suitable for the distributed storage and processing.
- Hadoop provides a command interface to interact with HDFS.
- The built-in servers of namenode and datanode help users to easily check the status of cluster.
- Streaming access to file system data.
- HDFS provides file permissions and authentication.

HDFS Architecture

Given below is the architecture of a Hadoop File System.



HDFS follows the master-slave architecture and it has the following elements.

Namenode

The namenode is the commodity hardware that contains the GNU/Linux operating system and the namenode software. It is a software that can be run on commodity hardware. The system having the namenode acts as the master server and it does the following tasks:

- Manages the file system namespace.
- Regulates client's access to files.
- It also executes file system operations such as renaming, closing, and opening files and directories.

Datanode

The datanode is a commodity hardware having the GNU/Linux operating system and datanode software. For every node Commodity hardware/System Commodity hardware/System in a cluster, there will be a datanode. These nodes manage the data storage of their system.

- Datanodes perform read-write operations on the file systems, as per client request.
- They also perform operations such as block creation, deletion, and replication according to the instructions of the namenode.

Block

Generally the user data is stored in the files of HDFS. The file in a file system will be divided into one or more segments and/or stored in individual data nodes. These file segments are called as blocks. In other words, the minimum amount of data that HDFS can read or write is called a Block. The default block size is 64MB, but it can be increased as per the need to change in HDFS configuration.

Goals of HDFS

- **Fault detection and recovery** : Since HDFS includes a large number of commodity hardware, failure of components is frequent. Therefore HDFS should have mechanisms for quick and automatic fault detection and recovery.
- **Huge datasets** : HDFS should have hundreds of nodes per cluster to manage the applications having huge datasets.
- **Hardware at data** : A requested task can be done efficiently, when the computation takes place near the data. Especially where huge datasets are involved, it reduces the network traffic and increases the throughput.

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?

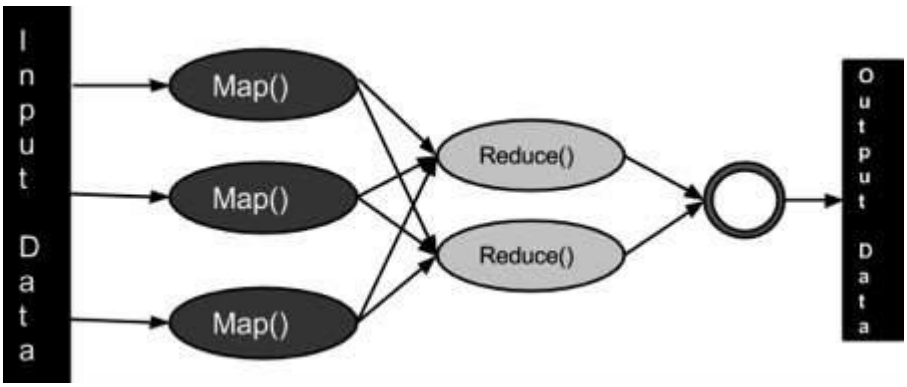
MapReduce is a processing technique and a program model for distributed computing based on java. The MapReduce algorithm contains two important tasks, namely Map and Reduce. Map takes a set of data and converts it into another set of data, where individual elements are broken down into tuples key/valuepairskey/valuepairs. 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 HDFSHDFS. 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 of MapReduce

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: InputInput <k1, v1> -> map -> <k2, v2>-> reduce -> <k3, v3>OutputOutput.

	Input	Output
Map	<k1, v1>	list <k2,v2><k2,v2>
Reduce	<k2, listv2v2>	list <k3,v3><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 HDFSHDFS.
- **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.

```
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(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/hadoope.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 http://mvnrepository.com/artifact/org.apache.hadoop/hadoop-core/1.2.1 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.

Options	Description
namenode -format	Formats the DFS filesystem.
secondarynamenode	Runs the DFS secondary namenode.
namenode	Runs the DFS namenode.
datanode	Runs a DFS datanode.
dfsadmin	Runs a DFS admin client.
mradmin	Runs a Map-Reduce admin client.
fsck	Runs a DFS filesystem checking utility.
fs	Runs a generic filesystem user client.
balancer	Runs a cluster balancing utility.
oiv	Applies the offline fsimage viewer to an fsimage.
fetchdt	Fetches a delegation token from the NameNode.
jobtracker	Runs the MapReduce job Tracker node.
pipes	Runs a Pipes job.
tasktracker	Runs a MapReduce task Tracker node.
historyserver	Runs job history servers as a standalone daemon.

job	Manipulates the MapReduce jobs.
queue	Gets information regarding JobQueues.
version	Prints the version.
jar <jar>	Runs a jar file.
distcp <srcurl> <desturl>	Copies file or directories recursively.
distcp2 <srcurl> <desturl>	DistCp version 2.
archive -archiveName NAME -p	Creates a hadoop archive.
<parent path> <src>* <dest>	
classpath	Prints the class path needed to get the Hadoop jar and the required libraries.
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.

GENERIC_OPTIONS	Description
-submit <job-file>	Submits the job.
-status <job-id>	Prints the map and reduce completion percentage and all job counters.
-counter <job-id> <group-name> <countername>	Prints the counter value.
-kill <job-id>	Kills the job.
-events <job-id> <fromevent-#> <#-of-events>	Prints the events' details received by jobtracker for the given range.
-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.

-list[all]	Displays all jobs. -list displays only jobs which are yet to complete.
-kill-task <task-id>	Kills the task. Killed tasks are NOT counted against failed attempts.
-fail-task <task-id>	Fails the task. Failed tasks are counted against failed attempts.
-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
```

Serialization and Deserialization in Hadoop

Serilaization is the process of converting structured objects into a byte stream. It is done basically for two purposes one, for transmission over a network(interprocess communication) and for writing to persisitent storage. In Hadoop the interprocess communication between nodes in the system is done by using remote procedure calls i.e. RPCs. The RPC rotocol uses serialization to make the message into a binary stream to be sent to the remote node,which receives and deserializes the binary stream into the original message.

RPC serialization format is expected to be:

- Compact: To efficenetly use network bandwidth.
- Fast: Very little performance overhead is expected for serialization and deserilization process.
- Extensible: To adept to new changes and reqirements.

- Interoperable:The format needs to be designed to support clients that are written in different languages to the server.

It should be noted that the data format for persistent storage purposes would have different requirements from serilaization framework in addition to four expected properties of an RPC's serialization format mentioned above.

- Compact : To efficenetly use storage space.
- Fast : To keep the overhead in reading or writing terabytes of data minimal.
- Extensible : To transparently read data written in older format.
- Interoperable :To read and write persistent using different languages.

Hadoop uses its own serialization format,Writables. Writable is compact and fast, but not extensible or interoperable.

The Writable Interface

The Writable interface has two methods, one for writing and one for reading. The method for writing writes its state to a DataOutput binary stream and the method for reading reads its state from a DataInput binary stream.

```
public interface Writable
{
void write(DataOutput out) throws IOException;
void readFields(DataOutput in)throws IOException;
}
```

Let us understand serialization with an example.Given below is a helper method.

```
public static byte[] serialize(Writable writable) throws IOException
 {
   ByteArrayOutputStream out = new ByteArrayOutputStream();
   DataOutputStream dataOut = new DataOutputStream(out);
   writable.write(dataOut);
   dataOut.close();
   return out.toByteArray();
}
```

Let's try deserialization. Again, we create a helper method to read a Writable object from a byte array:

```
public static byte[] deserialize(Writable writable, byte[] bytes) throws IOException
{
ByteArrayInputStream in = new ByteArrayInputStream(bytes);
DataInputStream dataIn = new DataInputStream(in);
writable.readFields(dataIn);
dataIn.close();
return bytes;
}
```

WritableComparable and comparators

IntWritable implements the WritableComparable interface, which is a subinterface of the Writable and java.lang.Comparable interfaces:

```
package org.apache.hadoop.io;
public interface WritableComparable extends Writable, Comparable

{
}
```

Comparison of types is important for MapReduce because in MapReduce there is sorting phase during which keys are compared with one another. Hadoop provides RawComparator extension of Java's Comparator:

```
package org.apache.hadoop.io;
import java.util.Comparator;
public interface RawComparator extends Comparator {
public int compare(byte[] b1,int s1,int l1,byte[] b2, int s2, int l2);
}
```

This interface permits implementors to compare records read from a stream without deserializing them into objects, hence avoiding any overhead of object creation. For example, the comparator for IntWritables implements the raw compare() method by reading an integer from each of the byte arrays b1 and b2 and comparing them directly from the given start positions (s1 and s2) and lengths (l1 and l2). WritableComparator is a general-purpose implementation of RawComparator for WritableComparable classes. It provides two main functions:

First, it provides a default implementation of the raw compare() method that deserializes the objects to be compared from the stream and invokes the object compare() method. Second, it acts as a factory for RawComparator instances (that Writable implementations have registered).

For example, to obtain a comparator for IntWritable, we just use: RawComparator comparator = WritableComparator.get(IntWritable.class); The comparator can be used to compare two IntWritable objects:

```
IntWritable w1 = new IntWritable(163);
IntWritable w2 = new IntWritable(67);
```

assertThat(comparator.compare(w1, w2), greaterThan(0)); or their serialized representations:

```
byte[] b1 = serialize(w1);
byte[] b2 = serialize(w2);
```

assertThat(comparator.compare(b1, 0, b1.length, b2, 0, b2.length), greaterThan(0));