HDFS cluster primarily consists of a **NameNode** that manages the file system **Metadata** and a **DataNodes** that stores the **actual data**.

* **NameNode:**NameNode can be considered as a master of the system. It maintains the file system tree and the metadata for all the files and directories present in the system. Two files **'Namespace image'** and the **'edit log'** are used to store metadata information. Namenode has knowledge of all the datanodes containing data blocks for a given file, however, it does not store block locations persistently. This information is reconstructed every time from datanodes when the system starts.
* **DataNode :**DataNodes are slaves which reside on each machine in a cluster and provide the actual storage. It is responsible for serving, read and write requests for the clients.

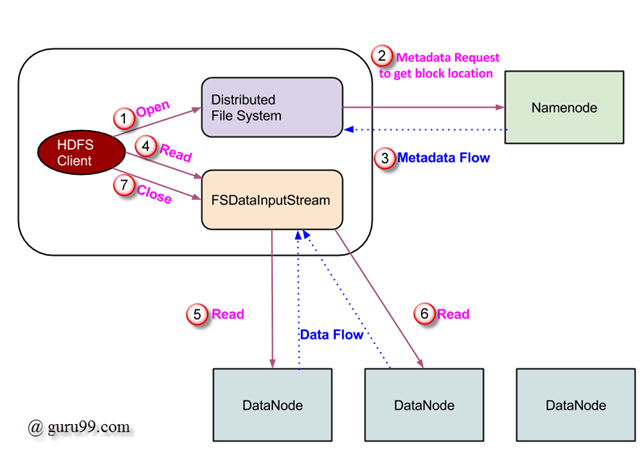
Read/write operations in HDFS operate at a block level. Data files in HDFS are broken into block-sized chunks, which are stored as independent units. Default block-size is 64 MB.

HDFS operates on a concept of data replication wherein multiple replicas of data blocks are created and are distributed on nodes throughout a cluster to enable high availability of data in the event of node failure.

***Do you know?***A file in HDFS, which is smaller than a single block, does not occupy a block's full storage.

**Read Operation In HDFS**

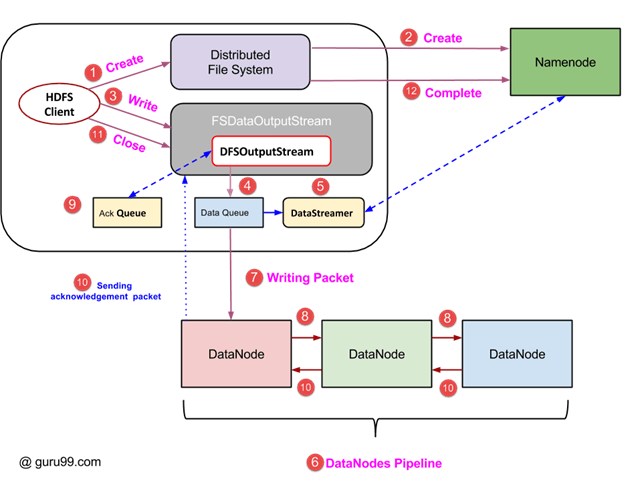
Data read request is served by HDFS, NameNode and DataNode. Let's call reader as a 'client'. Below diagram depicts file read operation in Hadoop.

[](http://cdn.guru99.com/images/Big_Data/061114_0923_LearnHDFSAB1.png)

1. Client initiates read request by calling **'open()'** method of FileSystem object; it is an object of type **DistributedFileSystem**.
2. This object connects to namenode using RPC and gets metadata information such as the locations of the blocks of the file. Please note that these addresses are of first few block of file.
3. In response to this metadata request, addresses of the DataNodes having copy of that block, is returned back.
4. Once addresses of DataNodes are received, an object of type **FSDataInputStream** is returned to the client. **FSDataInputStream** contains **DFSInputStream** which takes care of interactions with DataNode and NameNode. In step 4 shown in above diagram, client invokes **'read()'** method which causes **DFSInputStream** to establish a connection with the first DataNode with the first block of file.
5. Data is read in the form of streams wherein client invokes **'read()'** method repeatedly. This process of **read()** operation continues till it reaches end of block.
6. Once end of block is reached, DFSInputStream closes the connection and moves on to locate the next DataNode for the next block
7. Once client has done with the reading, it calls **close()** method.

**Write Operation In HDFS**

In this section, we will understand how data is written into HDFS through files.

[](http://cdn.guru99.com/images/Big_Data/061114_0923_LearnHDFSAB2.png)

1. Client initiates write operation by calling 'create()' method of DistributedFileSystem object which creates a new file - Step no. 1 in above diagram.
2. DistributedFileSystem object connects to the NameNode using RPC call and initiates new file creation. However, this file create operation does not associate any blocks with the file. It is the responsibility of NameNode to verify that the file (which is being created) does not exist already and client has correct permissions to create new file. If file already exists or client does not have sufficient permission to create a new file, then **IOException** is thrown to client. Otherwise, operation succeeds and a new record for the file is created by the NameNode.
3. Once new record in NameNode is created, an object of type FSDataOutputStream is returned to the client. Client uses it to write data into the HDFS. Data write method is invoked (step 3 in diagram).
4. FSDataOutputStream contains DFSOutputStream object which looks after communication with DataNodes and NameNode. While client continues writing data, **DFSOutputStream** continues creating packets with this data. These packets are en-queued into a queue which is called as **DataQueue**.
5. There is one more component called **DataStreamer** which consumes this **DataQueue**. DataStreamer also asks NameNode for allocation of new blocks thereby picking desirable DataNodes to be used for replication.
6. Now, the process of replication starts by creating a pipeline using DataNodes. In our case, we have chosen replication level of 3 and hence there are 3 DataNodes in the pipeline.
7. The DataStreamer pours packets into the first DataNode in the pipeline.
8. Every DataNode in a pipeline stores packet received by it and forwards the same to the second DataNode in pipeline.
9. Another queue, 'Ack Queue' is maintained by DFSOutputStream to store packets which are waiting for acknowledgement from DataNodes.
10. Once acknowledgement for a packet in queue is received from all DataNodes in the pipeline, it is removed from the 'Ack Queue'. In the event of any DataNode failure, packets from this queue are used to reinitiate the operation.
11. After client is done with the writing data, it calls close() method (Step 9 in the diagram) Call to close(), results into flushing remaining data packets to the pipeline followed by waiting for acknowledgement.
12. Once final acknowledgement is received, NameNode is contacted to tell it that the file write operation is complete.

**Access HDFS using JAVA API**

In this section, we try to understand Java interface used for accessing Hadoop's file system.

In order to interact with Hadoop's filesytem programmatically, Hadoop provides multiple JAVA classes. Package named org.apache.hadoop.fs contains classes useful in manipulation of a file in Hadoop's filesystem. These operations include, open, read, write, and close. Actually, file API for Hadoop is generic and can be extended to interact with other filesystems other than HDFS.

**Reading a file from HDFS, programmatically**

**Object java.net.URL** is used for reading contents of a file. To begin with, we need to make Java recognize Hadoop's hdfs URL scheme. This is done by calling **setURLStreamHandlerFactory** method on URL object and an instance of FsUrlStreamHandlerFactory is passed to it. This method needs to be executed only once per JVM, hence it is enclosed in a static block.

An example code is-

public class URLCat {

static {

URL.setURLStreamHandlerFactory(new FsUrlStreamHandlerFactory());

}

public static void main(String[] args) throws Exception {

InputStream in = null;

try {

in = new URL(args[0]).openStream();

IOUtils.copyBytes(in, System.out, 4096, false);

} finally {

IOUtils.closeStream(in);

}

}

}

This code opens and reads contents of a file. Path of this file on HDFS is passed to the program as a commandline argument.

**Access HDFS Using COMMAND-LINE INTERFACE**

This is one of the simplest way to interact with HDFS. Command-line interface has support for filesystem operations like read file, create directories, moving files, deleting data, and listing directories.

We can run **'$HADOOP\_HOME/bin/hdfs dfs -help'** to get detailed help on every command. Here, **'dfs'** is a shell command of HDFS which supports multiple subcommands.

Some of the widely used commands are listed below along with some details of each one.

1. Copy a file from local filesystem to HDFS

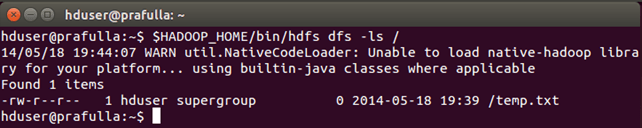
**$HADOOP\_HOME/bin/hdfs dfs -copyFromLocal temp.txt /**

[HDFS Tutorial: Read & Write  Commands using Java API](http://cdn.guru99.com/images/Big_Data/061114_0923_LearnHDFSAB3.png)

This command copies file temp.txt from local filesystem to HDFS.

2. We can list files present in a directory using **-ls**

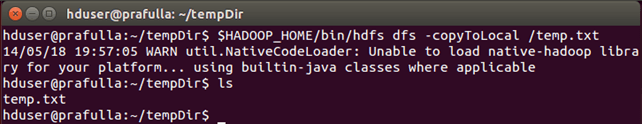
**$HADOOP\_HOME/bin/hdfs dfs -ls /**

[](http://cdn.guru99.com/images/Big_Data/061114_0923_LearnHDFSAB4.png)

We can see a file **'temp.txt'** (copied earlier) being listed under **' / '** directory.

3. Command to copy a file to local filesystem from HDFS

**$HADOOP\_HOME/bin/hdfs dfs -copyToLocal /temp.txt**

[](http://cdn.guru99.com/images/Big_Data/061114_0923_LearnHDFSAB5.png)

We can see **temp.txt** copied to local filesystem.

4. Command to create new directory

**$HADOOP\_HOME/bin/hdfs dfs -mkdir /mydirectory**

[HDFS Tutorial: Read & Write  Commands using Java API](http://cdn.guru99.com/images/Big_Data/061114_0923_LearnHDFSAB6.png)

Check whether directory is created or not. Now, you should know how to do it ;-)

# What is MapReduce? How it Works

MapReduce is a programming model suitable for processing of huge data. Hadoop is capable of running MapReduce programs written in various languages: Java, Ruby, Python, and C++. MapReduce programs are parallel in nature, thus are very useful for performing large-scale data analysis using multiple machines in the cluster.

**MapReduce programs work in two phases:**

1. Map phase
2. Reduce phase.

Input to each phase are **key-value** pairs. In addition, every programmer needs to specify two functions: **map function** and **reduce function**.

The whole process goes through three phase of execution namely,

**How MapReduce works**

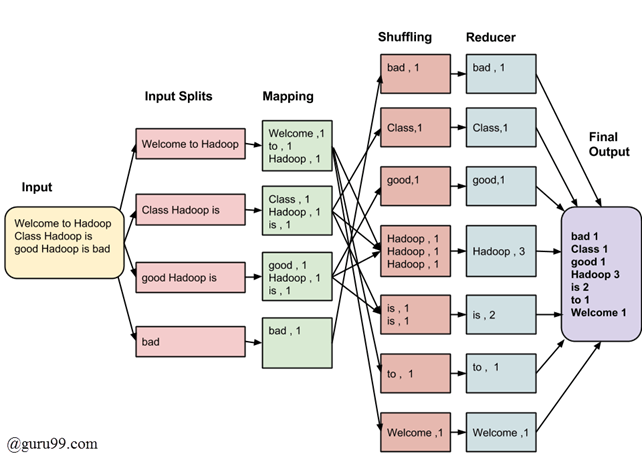
Lets understand this with an example –

Consider you have following input data for your MapReduce Program

Welcome to Hadoop Class

Hadoop is good

Hadoop is bad

[](http://cdn.guru99.com/images/Big_Data/061114_0930_Introductio1.png)

The final output of the MapReduce task is

|  |  |
| --- | --- |
| bad | 1 |
| Class | 1 |
| good | 1 |
| Hadoop | 3 |
| is | 2 |
| to | 1 |
| Welcome | 1 |

The data goes through following phases

**Input Splits:**

Input to a MapReduce job is divided into fixed-size pieces called **input splits**Input split is a chunk of the input that is consumed by a single map

**Mapping**

This is very first phase in the execution of map-reduce program. In this phase data in each split is passed to a mapping function to produce output values. In our example, job of mapping phase is to count number of occurrences of each word from input splits (more details about input-split is given below) and prepare a list in the form of <word, frequency>

**Shuffling**

This phase consumes output of Mapping phase. Its task is to consolidate the relevant records from Mapping phase output. In our example, same words are clubed together along with their respective frequency.

**Reducing**

In this phase, output values from Shuffling phase are aggregated. This phase combines values from Shuffling phase and returns a single output value. In short, this phase summarizes the complete dataset.

In our example, this phase aggregates the values from Shuffling phase i.e., calculates total occurrences of each words.

**The overall process in detail**

* One map task is created for each split which then executes map function for each record in the split.
* It is always beneficial to have multiple splits, because time taken to process a split is small as compared to the time taken for processing of the whole input. When the splits are smaller, the processing is better load balanced since we are processing the splits in parallel.
* However, it is also not desirable to have splits too small in size. When splits are too small, the overload of managing the splits and map task creation begins to dominate the total job execution time.
* For most jobs, it is better to make split size equal to the size of an HDFS block (which is 64 MB, by default).
* Execution of map tasks results into writing output to a local disk on the respective node and not to HDFS.
* Reason for choosing local disk over HDFS is, to avoid replication which takes place in case of HDFS store operation.
* Map output is intermediate output which is processed by reduce tasks to produce the final output.
* Once the job is complete, the map output can be thrown away. So, storing it in HDFS with replication becomes overkill.
* In the event of node failure before the map output is consumed by the reduce task, Hadoop reruns the map task on another node and re-creates the map output.
* Reduce task don't work on the concept of data locality. Output of every map task is fed to the reduce task. Map output is transferred to the machine where reduce task is running.
* On this machine the output is merged and then passed to the user defined reduce function.
* Unlike to the map output, reduce output is stored in HDFS (the first replica is stored on the local node and other replicas are stored on off-rack nodes). So, writing the reduce output

**How MapReduce Organizes Work?**

Hadoop divides the job into tasks. There are two types of tasks:

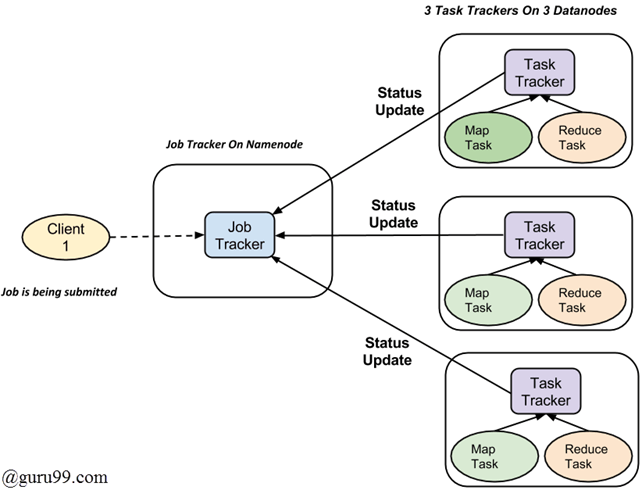
1. **Map tasks** (Spilts & Mapping)
2. **Reduce tasks** (Shuffling, Reducing)

as mentioned above.

The complete execution process (execution of Map and Reduce tasks, both) is controlled by two types of entities called a

1. **Jobtracker** : Acts like a **master** (responsible for complete execution of submitted job)
2. **Multiple Task Trackers** : Acts like **slaves,** each of them performing the job

For every job submitted for execution in the system, there is one **Jobtracker** that resides on **Namenode** and there are **multiple tasktrackers** which reside on **Datanode**.

[](http://cdn.guru99.com/images/Big_Data/061114_0930_Introductio2.png)

* A job is divided into multiple tasks which are then run onto multiple data nodes in a cluster.
* It is the responsibility of jobtracker to coordinate the activity by scheduling tasks to run on different data nodes.
* Execution of individual task is then look after by tasktracker, which resides on every data node executing part of the job.
* Tasktracker's responsibility is to send the progress report to the jobtracker.
* In addition, tasktracker periodically sends **'heartbeat'** signal to the Jobtracker so as to notify him of current state of the system.
* Thus jobtracker keeps track of overall progress of each job. In the event of task failure, the jobtracker can reschedule it on a different tasktracker.

# Hadoop & Mapreduce Examples: Create your First Program

**Problem Statement:**

Find out Number of Products Sold in Each Country.

[SalesJan2009.csv](https://drive.google.com/uc?export=download&id=0B_vqvT0ovzHcekp1WkVfUVNEdVE)

**Prerequisites:**

* This tutorial is developed on [**Linux - Ubuntu**](http://www.guru99.com/ubuntu-installation-on-virtual-box.html) operating System.
* You should have **Hadoop** (**version 2.2.0** used for this tutorial) already installed.
* You should have [**Java**](http://www.guru99.com/how-to-install-java-on-ubuntu.html) (**version 1.8.0** used for this tutorial) already installed on the system.

Before we start with the actual process, change user to 'hduser' (user used for Hadoop ).

**su - hduser\_**

[Hadoop & Mapreduce Examples: Create your First Program](http://cdn.guru99.com/images/Big_Data/061114_0954_CreateYourF1.png)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Transaction date** | **Product** | **Price** | **Payment Type** | **Name** | **City** | **State** | **Country** | **Account Created** | **Last Login** | **Latitude** | **Longitude** |
| 01-02-2009 6:17 | Product1 | 1200 | Mastercard | carolina | Basildon | England | United Kingdom | 01-02-2009 6:00 | 01-02-2009 6:08 | 51.5 | -1.1166667 |
| 01-02-2009 4:53 | Product1 | 1200 | Visa | Betina | Parkville | MO | United States | 01-02-2009 4:42 | 01-02-2009 7:49 | 39.195 | -94.68194 |
| 01-02-2009 13:08 | Product1 | 1200 | Mastercard | Federica e Andrea | Astoria | OR | United States | 01-01-2009 16:21 | 01-03-2009 12:32 | 46.18806 | -123.83 |
| 01-03-2009 14:44 | Product1 | 1200 | Visa | Gouya | Echuca | Victoria | Australia | 9/25/05 21:13 | 01-03-2009 14:22 | -36.1333333 | 144.75 |
| 01-04-2009 12:56 | Product2 | 3600 | Visa | Gerd W | Cahaba Heights | AL | United States | 11/15/08 15:47 | 01-04-2009 12:45 | 33.52056 | -86.8025 |
| 01-04-2009 13:19 | Product1 | 1200 | Visa | LAURENCE | Mickleton | NJ | United States | 9/24/08 15:19 | 01-04-2009 13:04 | 39.79 | -75.23806 |

#### Steps: 1

Create a new directory with name **MapReduceTutorial**

**sudo mkdir MapReduceTutorial**

[Hadoop & Mapreduce Examples: Create your First Program](http://cdn.guru99.com/images/Big_Data/061114_0954_CreateYourF2.png)

**Give permissions**

**sudo chmod -R 777 MapReduceTutorial**

[Hadoop & Mapreduce Examples: Create your First Program](http://cdn.guru99.com/images/Big_Data/061114_0954_CreateYourF3.png)**SalesMapper.java**

package SalesCountry;

import java.io.IOException;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.LongWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapred.\*;

public class SalesMapper extends MapReduceBase implements Mapper <LongWritable, Text, Text, IntWritable> {

private final static IntWritable one = new IntWritable(1);

public void map(LongWritable key, Text value, OutputCollector <Text, IntWritable> output, Reporter reporter) throws IOException {

String valueString = value.toString();

String[] SingleCountryData = valueString.split(",");

output.collect(new Text(SingleCountryData[7]), one);

}

}

**SalesCountryReducer.java**

package SalesCountry;

import java.io.IOException;

import java.util.\*;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapred.\*;

public class SalesCountryReducer extends MapReduceBase implements Reducer<Text, IntWritable, Text, IntWritable> {

public void reduce(Text t\_key, Iterator<IntWritable> values, OutputCollector<Text,IntWritable> output, Reporter reporter) throws IOException {

Text key = t\_key;

int frequencyForCountry = 0;

while (values.hasNext()) {

// replace type of value with the actual type of our value

IntWritable value = (IntWritable) values.next();

frequencyForCountry += value.get();

}

output.collect(key, new IntWritable(frequencyForCountry));

}

}

**SalesCountryDriver.java**

package SalesCountry;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.\*;

import org.apache.hadoop.mapred.\*;

public class SalesCountryDriver {

public static void main(String[] args) {

JobClient my\_client = new JobClient();

// Create a configuration object for the job

JobConf job\_conf = new JobConf(SalesCountryDriver.class);

// Set a name of the Job

job\_conf.setJobName("SalePerCountry");

// Specify data type of output key and value

job\_conf.setOutputKeyClass(Text.class);

job\_conf.setOutputValueClass(IntWritable.class);

// Specify names of Mapper and Reducer Class

job\_conf.setMapperClass(SalesCountry.SalesMapper.class);

job\_conf.setReducerClass(SalesCountry.SalesCountryReducer.class);

// Specify formats of the data type of Input and output

job\_conf.setInputFormat(TextInputFormat.class);

job\_conf.setOutputFormat(TextOutputFormat.class);

// Set input and output directories using command line arguments,

//arg[0] = name of input directory on HDFS, and arg[1] = name of output directory to be created to store the output file.

FileInputFormat.setInputPaths(job\_conf, new Path(args[0]));

FileOutputFormat.setOutputPath(job\_conf, new Path(args[1]));

my\_client.setConf(job\_conf);

try {

// Run the job

JobClient.runJob(job\_conf);

} catch (Exception e) {

e.printStackTrace();

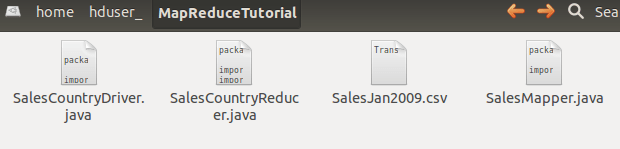
}

}

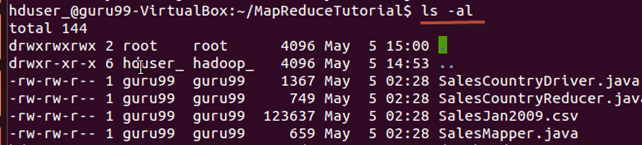
}

[Download Files Here](https://drive.google.com/uc?export=download&id=0B_vqvT0ovzHcekp1WkVfUVNEdVE)

If you want to understand the code in these files refer this [Guide](http://www.guru99.com/understanding-map-reducer-code.html)

[](http://cdn.guru99.com/images/Big_Data/061114_0954_CreateYourF4.png)

Check the file permissions of all these files

[](http://cdn.guru99.com/images/Big_Data/061114_0954_CreateYourF5.png)

and if 'read' permissions are missing then grant the same-

[Hadoop & Mapreduce Examples: Create your First Program](http://cdn.guru99.com/images/Big_Data/061114_0954_CreateYourF6.png)

#### Steps: 2

Export classpath

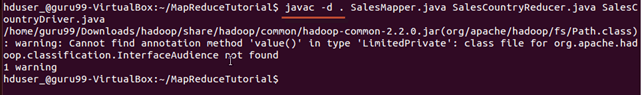
export CLASSPATH="$HADOOP\_HOME/share/hadoop/mapreduce/hadoop-mapreduce-client-core-2.2.0.jar:$HADOOP\_HOME/share/hadoop/mapreduce/hadoop-mapreduce-client-common-2.2.0.jar:$HADOOP\_HOME/share/hadoop/common/hadoop-common-2.2.0.jar:~/MapReduceTutorial/SalesCountry/\*:$HADOOP\_HOME/lib/\*"

[Hadoop & Mapreduce Examples: Create your First Program](http://cdn.guru99.com/images/Big_Data/061114_0954_CreateYourF7.png)

#### Steps: 3

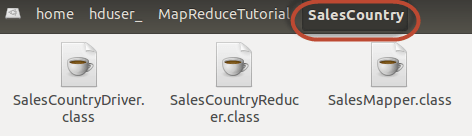
Compile java files (these files are present in directory **Final-MapReduceHandsOn**). Its class files will be put in the package directory

javac -d . SalesMapper.java SalesCountryReducer.java SalesCountryDriver.java

[](http://cdn.guru99.com/images/Big_Data/061114_0954_CreateYourF8.png)

**This warning can be safely ignored.**

This compilation will create a directory in a current directory named with package name specified in the java source file (i.e. **SalesCountry** in our case) and put all compiled class files in it.

[](http://cdn.guru99.com/images/Big_Data/061114_0954_CreateYourF9.png)

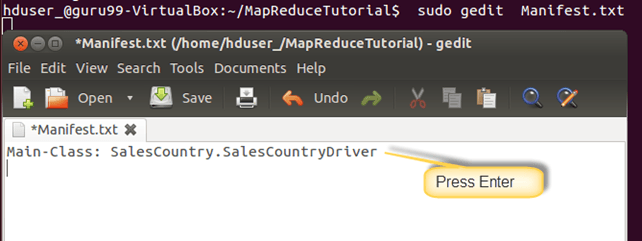
#### Steps: 4

Create a new file **Manifest.txt**

**sudo gedit Manifest.txt**

add following lines to it,

Main-Class: SalesCountry.SalesCountryDriver

[](http://cdn.guru99.com/images/Big_Data/061114_0954_CreateYourF10.png)

**SalesCountry.SalesCountryDriver** is name of main class. Please note that you have to hit enter key at end of this line.

#### Steps: 5

Create a Jar file

jar cfm ProductSalePerCountry.jar Manifest.txt SalesCountry/\*.class

[Hadoop & Mapreduce Examples: Create your First Program](http://cdn.guru99.com/images/Big_Data/061114_0954_CreateYourF11.png)

Check that the jar file is created

[Hadoop & Mapreduce Examples: Create your First Program](http://cdn.guru99.com/images/Big_Data/061114_0954_CreateYourF12.png)

#### Steps: 6

Start Hadoop

$HADOOP\_HOME/sbin/start-dfs.sh

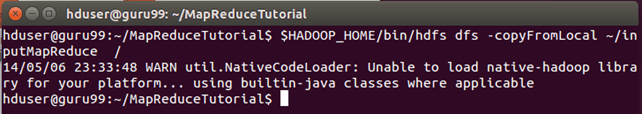
$HADOOP\_HOME/sbin/start-yarn.sh

#### Steps: 7

Copy the File **SalesJan2009.csv** into **~/inputMapReduce**

Now Use below command to copy **~/inputMapReduce**to HDFS.

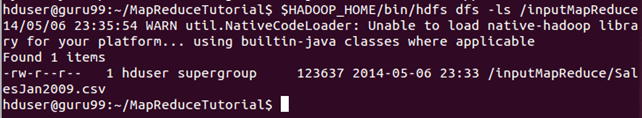
$HADOOP\_HOME/bin/hdfs dfs -copyFromLocal ~/inputMapReduce /

[](http://cdn.guru99.com/images/Big_Data/061114_0954_CreateYourF13.png)

We can safely ignore this warning.

Verify whether file is actually copied or not.

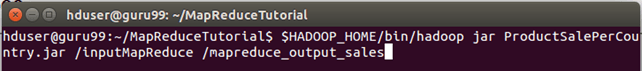
$HADOOP\_HOME/bin/hdfs dfs -ls /inputMapReduce

[](http://cdn.guru99.com/images/Big_Data/061114_0954_CreateYourF14.png)

#### Steps: 8

Run MapReduce job

$HADOOP\_HOME/bin/hadoop jar ProductSalePerCountry.jar /inputMapReduce /mapreduce\_output\_sales

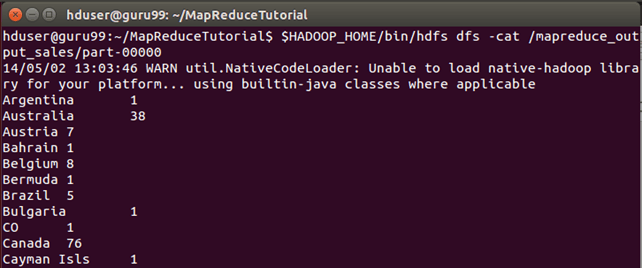
[](http://cdn.guru99.com/images/Big_Data/061114_0954_CreateYourF15.png)

This will create an output directory named mapreduce\_output\_sales on HDFS. Contents of this directory will be a file containing product sales per country.

#### Steps: 9

Result can be seen through command interface as,

$HADOOP\_HOME/bin/hdfs dfs -cat /mapreduce\_output\_sales/part-00000

[](http://cdn.guru99.com/images/Big_Data/061114_0954_CreateYourF16.png)

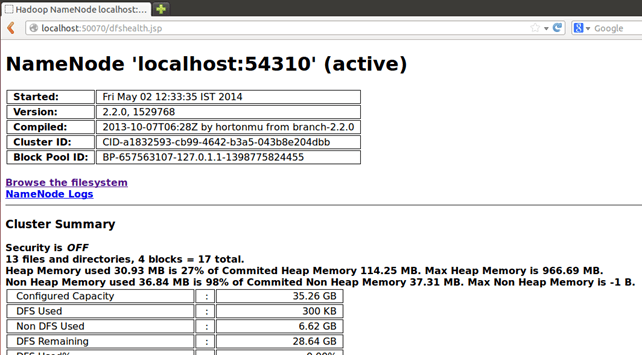
**o/p of above**

**OR**

Results can also be seen via web interface as-

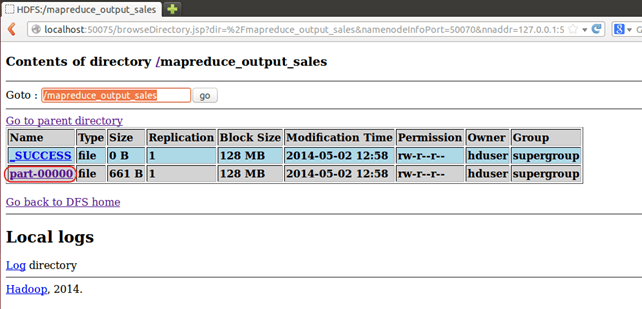
**Results through web interface-**

Open r in web browser.

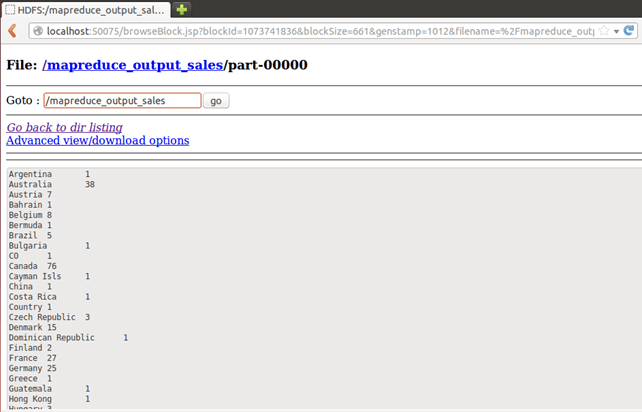
[](http://cdn.guru99.com/images/Big_Data/061114_0954_CreateYourF17.png)

Now select **'Browse the filesystem'**and navigate upto **/mapreduce\_output\_sales**

**o/p of above**

[](http://cdn.guru99.com/images/Big_Data/061114_0954_CreateYourF18.png)

Open **part-r-00000**

[](http://cdn.guru99.com/images/Big_Data/061114_0954_CreateYourF19.png)

### Understanding MapReducer Code

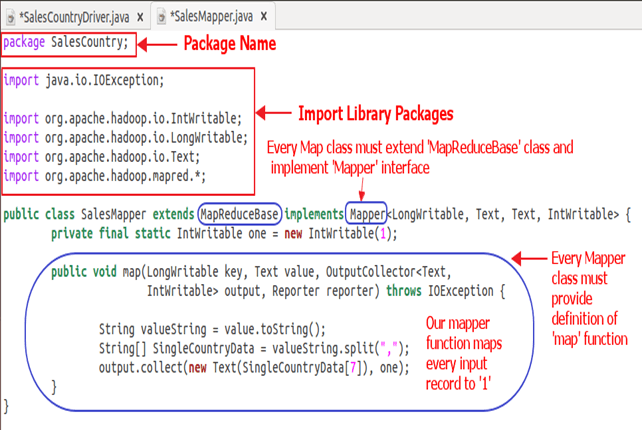
#### Explanation of SalesMapper Class

In this section we will understand implementation of **SalesMapper** class.

1. We begin by specifying name of package for our class. **SalesCountry** is name of out package. Please note that output of compilation, **SalesMapper.class** will go into directory named by this package name: **SalesCountry**.

Followed by this, we import library packages.

Below snapshot shows implementation of **SalesMapper** class-

[](http://cdn.guru99.com/images/Big_Data/061114_0959_Understandi1.png)

Code Explanation:

**1. SalesMapper Class Definition-**

public class SalesMapper extends MapReduceBase implements Mapper<LongWritable, Text, Text, IntWritable> {

Every mapper class must be extended from **MapReduceBase** class and it must implement **Mapper** interface.

**2. Defining 'map' function-**

public void map(LongWritable key,

Text value,

OutputCollector<Text, IntWritable> output,

Reporter reporter) throws IOException

Main part of Mapper class is a **'map()'** method which accepts four arguments.

At every call to **'map()'** method, a **key-value** pair (**'key'** and **'value'** in this code) is passed.

**'map()'** method begins by splitting input text which is received as an argument. It uses tokenizer to split these lines into words.

String valueString = value.toString();

String[] SingleCountryData = valueString.split(",");

Here, **','** is used as a delimiter.

After this, a pair is formed using a record at 7th index of array **'SingleCountryData'** and a value **'1'**.

        output.collect(new Text(SingleCountryData[7]), one);

We are choosing record at 7th index because we need **Country** data and it is located at 7th index in array **'SingleCountryData'**.

Please note that our input data is in the below format (where **Country** is at 7th index, with 0 as a starting index)-

Transaction\_date,Product,Price,Payment\_Type,Name,City,State,**Country**,Account\_Created,Last\_Login,Latitude,Longitude

Output of mapper is again a **key-value** pair which is outputted using **'collect()'** method of **'OutputCollector'**.

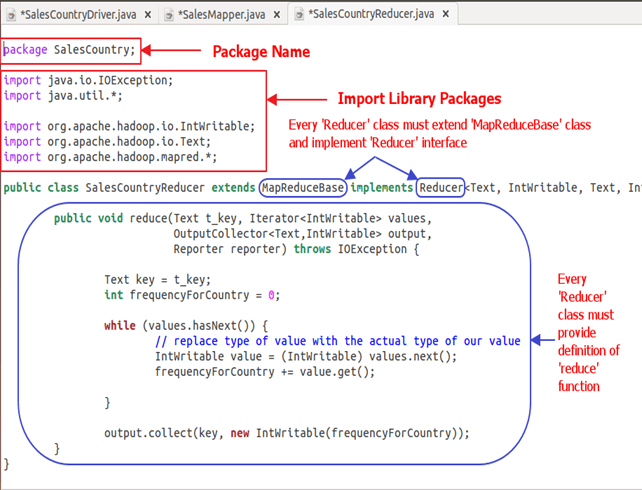
### Explanation of SalesCountryReducer Class

In this section we will understand implementation of **SalesCountryReducer** class.

1. We begin by specifying name of package for our class. **SalesCountry** is name of out package. Please note that output of compilation, **SalesCountryReducer.class** will go into directory named by this package name: **SalesCountry**.

Followed by this, we import library packages.

Below snapshot shows implementation of **SalesCountryReducer** class-

[](http://cdn.guru99.com/images/Big_Data/061114_0959_Understandi2.png)

Code Explanation:

**1. SalesCountryReducer Class Definition-**

public class SalesCountryReducer extends MapReduceBase implements Reducer<Text, IntWritable, Text, IntWritable> {

Here, first two data types, **'Text'** and **'IntWritable'** are data type of input key-value to the reducer.

Output of mapper is in the form of <CountryName1, 1>, <CountryName2, 1>. This output of mapper becomes input to the reducer. So, to align with its data type, **Text** and **IntWritable** are used as data type here.

The last two data types, 'Text' and 'IntWritable' are data type of output generated by reducer in the form of key-value pair.

Every reducer class must be extended from **MapReduceBase** class and it must implement **Reducer** interface.

**2. Defining 'reduce' function-**

public void reduce( Text t\_key,

Iterator<IntWritable> values,

OutputCollector<Text,IntWritable> output,

Reporter reporter) throws IOException {

Input to the **reduce()** method is a key with list of multiple values.

For example, in our case it will be-

<United Arab Emirates, 1>, <United Arab Emirates, 1>, <United Arab Emirates, 1>,<United Arab Emirates, 1>, <United Arab Emirates, 1>, <United Arab Emirates, 1>.

This is given to reducer as **<United Arab Emirates, {1,1,1,1,1,1}>**

So, to accept arguments of this form, first two data types are used, viz., **Text** and **Iterator<IntWritable>**. **Text** is a data type of key and **Iterator<IntWritable>** is a data type for list of values for that key.

The next argument is of type **OutputCollector<Text,IntWritable>** which collects output of reducer phase.

**reduce()** method begins by copying key value and initializing frequency count to 0.

        Text key = t\_key;  
        int frequencyForCountry = 0;

Then, using '**while'** loop, we iterate through the list of values associated with the key and calculate the final frequency by summing up all the values.

while (values.hasNext()) {

// replace type of value with the actual type of our value

IntWritable value = (IntWritable) values.next();

frequencyForCountry += value.get();

}

Now, we push the result to the output collector in the form of **key** and obtained **frequency count**.

Below code does this-

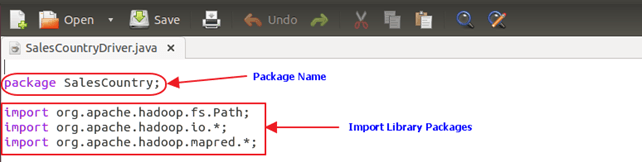
        output.collect(key, new IntWritable(frequencyForCountry));

### Explanation of SalesCountryDriver Class

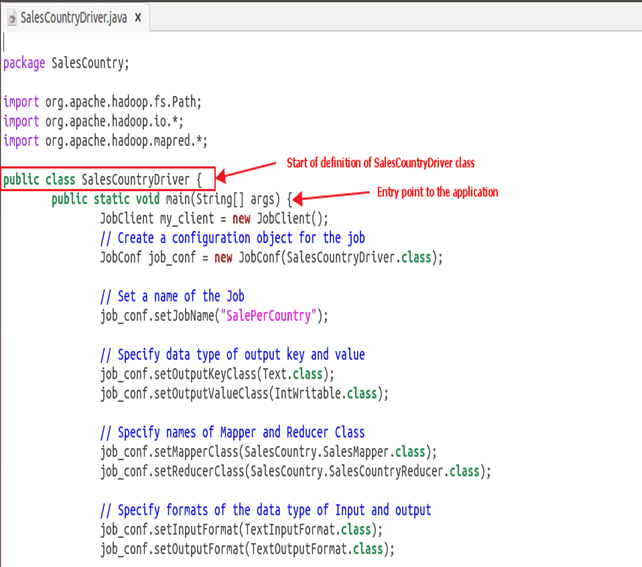
In this section we will understand implementation of **SalesCountryDriver** class

1. We begin by specifying name of package for our class. **SalesCountry** is name of out package. Please note that output of compilation, **SalesCountryDriver.class** will go into directory named by this package name: **SalesCountry**.

Here is a line specifying package name followed by code to import library packages.

[](http://cdn.guru99.com/images/Big_Data/061114_0959_Understandi3.png)

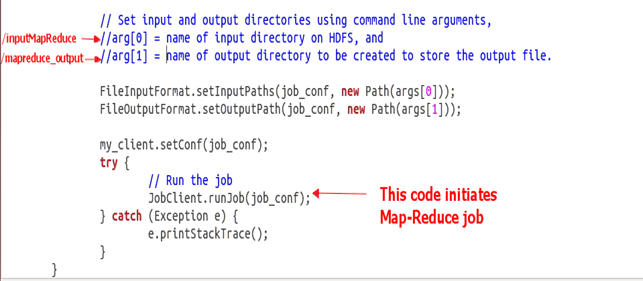
2. Define a driver class which will create a new client job, configuration object and advertise Mapper and Reducer classes.

The driver class is responsible for setting our MapReduce job to run in Hadoop. In this class, we specify **job name, data type of input/output and names of mapper and reducer classes**.[](http://cdn.guru99.com/images/Big_Data/061114_0959_Understandi4.png)

3. In below code snippet, we set input and output directories which are used to consume input dataset and produce output, respectively.

**arg[0]** and **arg[1]** are the command-line arguments passed with a command given in MapReduce hands-on, i.e.,

**$HADOOP\_HOME/bin/hadoop jar ProductSalePerCountry.jar /inputMapReduce /mapreduce\_output\_sales**

[](http://cdn.guru99.com/images/Big_Data/061114_0959_Understandi5.png)

4. Trigger our job

Below code start execution of MapReduce job-

try {

// Run the job

JobClient.runJob(job\_conf);

} catch (Exception e) {

e.printStackTrace();

}