



***MAP REDUCE DEMONSTRATION***

*Submitted in partial fulfilment for the Subject*

*BIG DATA LAB*

IN

**INFORMATION SCIENCE AND ENGINEERING**

BY

N ADITYA NAYAK **1NT17IS099**

PAWAN KUMAR **1NT17IS121**

**Under the Guidance of**

**PRASHANT B.S**

Professor

Information Science and Engineering

NMIT, Bengaluru.

**DEPARTMENT OF INFORMATION SCIENCE AND ENGINEERING**

(Accredited by NBA Tier-1)

2020

**INTRODUCTION TO HADOOP**

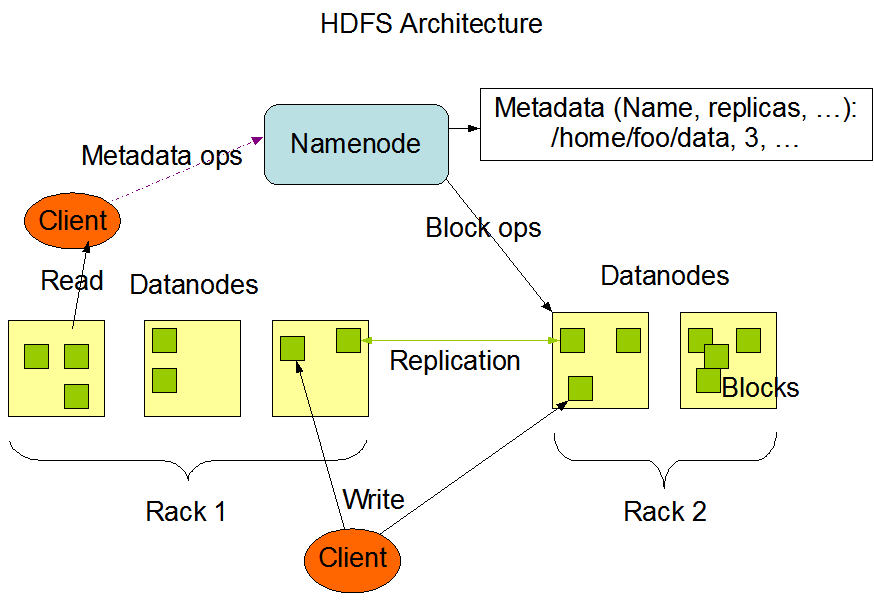
Hadoop is an open-source software framework for storing data and running applications on clusters of commodity hardware. It provides massive storage for any kind of data, enormous processing power and the ability to handle virtually limitless concurrent tasks or jobs**.**

The core of Apache Hadoop consists of a storage part, known as Hadoop Distributed File System (HDFS), and a processing part which is a MapReduce programming model. Hadoop splits files into large blocks and distributes them across nodes in a cluster. It then transfers packaged code into nodes to process the data in parallel. This approach takes advantage of datalocality where nodes manipulate the data they have access to. This allows the dataset to be processed faster and more efficiently than it would be in a more conventional SUPER COMPUTER ARCHITECTURE that relies on a parallel file [s](https://en.wikipedia.org/wiki/Parallel_file_system)ystem where computation and data are distributed via high-speed networking.

The base Apache Hadoop framework is composed of the following modules:

* ***Hadoop Common*** – contains libraries and utilities needed by other Hadoop modules;
* ***Hadoop Distributed File System (HDFS)*** – a distributed file-system that stores data on commodity machines, providing very high aggregate bandwidth across the cluster;
* ***Hadoop YARN*** – (introduced in 2012) a platform responsible for managing computing resources in clusters and using them for scheduling users' applications;
* ***Hadoop MapReduce*** – an implementation of the MapReduce programming model for large-scale data processing.

**HDFS ARCHITECTURE**



The Hadoop Distributed File System (HDFS) is a distributed file system designed to run on commodity hardware. It has many similarities with existing distributed file systems. However, the differences from other distributed file systems are significant. HDFS is highly fault-tolerant and is designed to be deployed on low-cost hardware. HDFS provides high throughput access to application data and is suitable for applications that have large data sets. HDFS relaxes a few POSIX requirements to enable streaming access to file system data. HDFS was originally built as infrastructure for the Apache Nutch web search engine project. HDFS is part of the Apache Hadoop Core project.

The NameNode and DataNode are pieces of software designed to run on commodity machines. These machines typically run a GNU/Linux operating system (OS). HDFS is built using the Java language; any machine that supports Java can run the NameNode or the DataNode software. Usage of the highly portable Java language means that HDFS can be deployed on a wide range of machines. A typical deployment has a dedicated machine that runs only the NameNode software. Each of the other machines in the cluster runs one instance of the DataNode software. The architecture does not preclude running multiple DataNodes on the same machine but in a real deployment that is rarely the case.

The existence of a single NameNode in a cluster greatly simplifies the architecture of the system. The NameNode is the arbitrator and repository for all HDFS metadata. The system is designed in such a way that user data never flows through the NameNode.

## The File System Namespace

HDFS supports a traditional hierarchical file organization. A user or an application can create directories and store files inside these directories. The file system namespace hierarchy is similar to most other existing file systems; one can create and remove files, move a file from one directory to another, or rename a file. HDFS supports quoats and permissions. HDFS does not support hard links or soft links. However, the HDFS architecture does not preclude implementing these features.

While HDFS follows nameing convention of file systems some paths and names (e.g. /.reserved and .snapshot ) are reserved.

The NameNode maintains the file system namespace. Any change to the file system namespace or its properties is recorded by the NameNode. An application can specify the number of replicas of a file that should be maintained by HDFS. The number of copies of a file is called the replication factor of that file. This information is stored by the NameNode.

## Data Replication

HDFS is designed to reliably store very large files across machines in a large cluster. It stores each file as a sequence of blocks. The blocks of a file are replicated for fault tolerance. The block size and replication factor are configurable per file.

All blocks in a file except the last block are the same size, while users can start a new block without filling out the last block to the configured block size after the support for variable length block was added to append and hsync.

An application can specify the number of replicas of a file. The replication factor can be specified at file creation time and can be changed later. Files in HDFS are write-once (except for appends and truncates) and have strictly one writer at any time.

The NameNode makes all decisions regarding replication of blocks. It periodically receives a Heartbeat and a Blockreport from each of the DataNodes in the cluster. Receipt of a Heartbeat implies that the DataNode is functioning properly. A Blockreport contains a list of all blocks on a DataNode.

**PROBLEM STATEMENT**

Problem Statement-I

Create a dataset in excel as .csv file and it should contain the following fields with at least 20 sample datasets in it.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| NAME | USN | S1 | S2 | S3 |
| AKUL | 1NT17IS067 | 70 | 80 | 90 |

Use the Hadoop MapReduce programming framework to come up with a Program which will take the data from this .csv file and computes the following

1. Total number of students who have scored more than 60 in Subject 1

2. Total number of students who have passed in all the subjects

**CODE**

package aaa;

import java.io.IOException;

import java.util.\*;

import org.apache.hadoop.fs.Path;

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

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

public class Count1115{

//MAPPER CODE

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

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

Text myText=new Text();

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

String myString = value.toString();

String[] csvArray = myString.split(",");

myText.set("mykey");

IntWritable amount =new IntWritable(Integer.parseInt(csvArray[2]));

if(amount.get()>60)

{

output.collect(myText,one);

}

}

}

//REDUCER CODE

public static class Reduce 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 { //{little: {1,1}}

int count = 0 ;

//String myString = key.toString();

Text key=t\_key;

while(values.hasNext()) {

IntWritable value = values.next();

count += value.get();

}

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

}

}

//DRIVER CODE

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

JobConf conf = new JobConf(Count1115.class);

conf.setJobName("countclass");

conf.setOutputKeyClass(Text.class);

conf.setOutputValueClass(IntWritable.class);

conf.setMapperClass(Map.class);

conf.setCombinerClass(Reduce.class);

conf.setReducerClass(Reduce.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);

}

}

**EXPLANATION**

**1)Mapper Function**

[Mapper](https://hadoop.apache.org/docs/current/api/org/apache/hadoop/mapreduce/Mapper.html) maps input key/value pairs to a set of intermediate key/value pairs.

Maps are the individual tasks that transform input records into intermediate records. The transformed intermediate records do not need to be of the same type as the input records. A given input pair may map to zero or many output pairs.

The Hadoop MapReduce framework spawns one map task for each InputSplit generated by the InputFormat for the job.

Overall, mapper implementations are passed to the job via [Job.setMapperClass(Class)](https://hadoop.apache.org/docs/current/api/org/apache/hadoop/mapreduce/Job.html) method. The framework then calls [map(WritableComparable, Writable, Context)](https://hadoop.apache.org/docs/current/api/org/apache/hadoop/mapreduce/Mapper.html) for each key/value pair in the InputSplit for that task. Applications can then override the cleanup(Context) method to perform any required cleanup.

Output pairs do not need to be of the same types as input pairs. A given input pair may map to zero or many output pairs. Output pairs are collected with calls to context.write(WritableComparable, Writable).

**2)Reducer Function**

[Reducer](https://hadoop.apache.org/docs/current/api/org/apache/hadoop/mapreduce/Reducer.html) reduces a set of intermediate values which share a key to a smaller set of values.

The number of reduces for the job is set by the user via [Job.setNumReduceTasks(int)](https://hadoop.apache.org/docs/current/api/org/apache/hadoop/mapreduce/Job.html).

Overall, Reducer implementations are passed the Job for the job via the [Job.setReducerClass(Class)](https://hadoop.apache.org/docs/current/api/org/apache/hadoop/mapreduce/Job.html) method and can override it to initialize themselves. The framework then calls [reduce(WritableComparable, Iterable<Writable>, Context)](https://hadoop.apache.org/docs/current/api/org/apache/hadoop/mapreduce/Reducer.html) method for each <key, (list of values)> pair in the grouped inputs. Applications can then override the cleanup(Context) method to perform any required cleanup.

Reducer has 3 primary phases: shuffle, sort and reduce.

**3)Driver Code**

There is one final component of a **Hadoop MapReduce program**, called the **Driver**. The **driver** initializes the job and instructs the **Hadoop** platform to execute your **code** on a set of input files, and controls where the output files are placed.

**STEPS TO EXCEUTE THIS PROGRAM**

After Starting the hadoop deamons,

1)Create an input directory

$ hdfs dfs -mkdir -p ~/input

2)Load the .csv file into the input folder

$ hdfs dfs -put /home/aditya/Desktop/data.csv ~/input/

3)Compile jar file (which is generated via ECLIPSE IDE)

$ hadoop jar /home/aditya/Desktop/p1.jar ~/input ~/output

**NOTE** There is no need to create output folder as its created by the hadoop itself.

4)To See The Output

$ hdfs dfs -cat ~/output/part\*