**DFS or Distributed File System:**

Distributed File System talks about **managing data**, i.e. **files or folders across multiple computers or servers.** In other words, DFS is a file system that allows us to store data over multiple nodes or machines in a cluster and allows multiple users to access data. So basically, it serves the same purpose as the file system which is available in your machine, like for windows you have NTFS (New Technology File System) or for Mac you have HFS (Hierarchical File System). The only difference is that, in case of Distributed File System, you store data in multiple machines rather than single machine. Even though the files are stored across the network, DFS organizes, and displays data in such a manner that a user sitting on a machine will feel like all the data is stored in that very machine.

## **What is HDFS?**

Hadoop is data divided into multiple chunks of data and store it and process the data very fast

Hadoop Distributed file system or HDFS is a Java based distributed file system that allows you to store large data across multiple nodes in a Hadoop cluster. So, if you install Hadoop, you get HDFS as an underlying storage system for storing the data in the distributed environment.

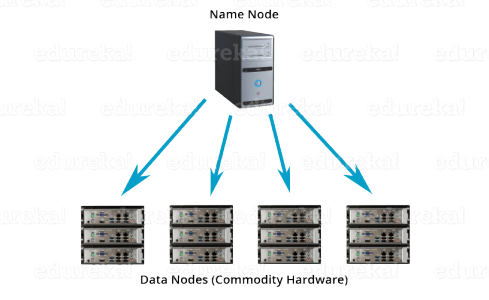
Let’s take an example to understand it. Imagine that you have ten machines or ten computers with a hard drive of 1 TB on each machine. Now, HDFS says that if you install Hadoop as a platform on top of these ten machines, you will get HDFS as a storage service. Hadoop Distributed File System is distributed in such a way that every machine contributes their individual storage for storing any kind of data.

**HDFS Tutorial: Features of HDFSs**

We will understand these features in detail when we will explore the HDFS Architecture in our next HDFS tutorial blog. But, for now, let’s have an overview on the features of HDFS:

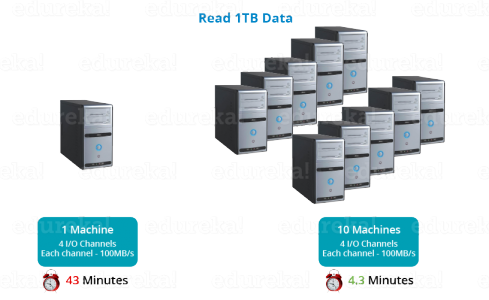
* ***Cost:*** The HDFS, in general, is deployed on a commodity hardware like your desktop/laptop which you use every day. So, it is very economical in terms of the cost of ownership of the project. Since, we are using low cost commodity hardware, you don’t need to spend huge amount of money for scaling out your Hadoop cluster. In other words, adding more nodes to your HDFS is cost effective.
* ***Variety and Volume of Data:*** When we talk about HDFS then we talk about storing huge data i.e. Terabytes & petabytes of data and different kinds of data. So, you can store any type of data into HDFS, be it structured, unstructured or semi structured.
* ***Reliability and Fault Tolerance:*** When you store data on HDFS, it internally divides the given data into data blocks and stores it in a distributed fashion across your Hadoop cluster. The information regarding which data block is located on which of the data nodes is recorded in the metadata. **NameNode** manages the meta data and the **DataNodes** are responsible for storing the data.  
  Name node also replicates the data i.e. maintains multiple copies of the data. This replication of the data makes HDFS very reliable and fault tolerant. So, even if any of the nodes fails, we can retrieve the data from the replicas residing on other data nodes. By default, the replication factor is 3. Therefore, if you store 1 GB of file in HDFS, it will finally occupy 3 GB of space. The name node periodically updates the metadata and maintains the replication factor consistent.
* ***Data Integrity:*** Data Integrity talks about whether the data stored in my HDFS are correct or not. HDFS constantly checks the integrity of data stored against its checksum. If it finds any fault, it reports to the name node about it. Then, the name node creates additional new replicas and therefore deletes the corrupted copies.
* ***High Throughput:*** Throughput is the amount of work done in a unit time. It talks about how fast you can access the data from the file system. Basically, it gives you an insight about the system performance. As you have seen in the above example where we used ten machines collectively to enhance computation. There we were able to reduce the processing time from **43 minutes** to a mere **4.3 minutes** as all the machines were working in parallel. Therefore, by processing data in parallel, we decreased the processing time tremendously and thus, achieved high throughput.
* ***Data Locality:***Data locality talks about moving processing unit to data rather than the data to processing unit. In our traditional system, we used to bring the data to the application layer and then process it. But now, because of the architecture and huge volume of the data, bringing the data to the application layer will reduce the network performance to a noticeable extent. So, in HDFS, we bring the computation part to the data nodes where the data is residing. Hence, you are not moving the data, you are bringing the program or processing part to the data.

### 1. Distributed Storage:



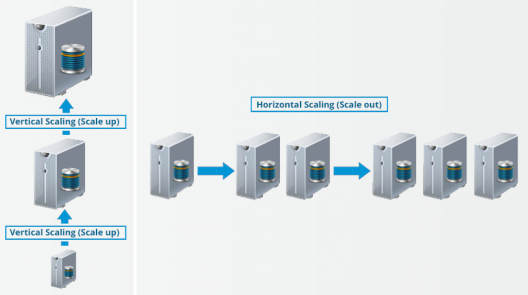
When you access Hadoop Distributed file system from any of the ten machines in the Hadoop cluster, you will feel as if you have logged into a single large machine which has a storage capacity of 10 TB (total storage over ten machines). What does it mean? It means that you can store a single large file of 10 TB which will be distributed over the ten machines (1 TB each). So, it is **not limited to the physical boundaries** of each individual machine.

### 2. Distributed & Parallel Computation:



Because the data is divided across the machines, it allows us to take advantage of **Distributed and Parallel Computation**. Let’s understand this concept by the above example. Suppose, it takes 43 minutes to process 1 TB file on a single machine. So, now tell me, how much time will it take to process the same 1 TB file when you have 10 machines in a Hadoop cluster with similar configuration – 43 minutes or 4.3 minutes? 4.3 minutes, Right! What happened here? Each of the nodes is working with a part of the 1 TB file in parallel. Therefore, the work which was taking 43 minutes before, gets finished in just 4.3 minutes now as the work got divided over ten machines.

### 3. Horizontal Scalability:

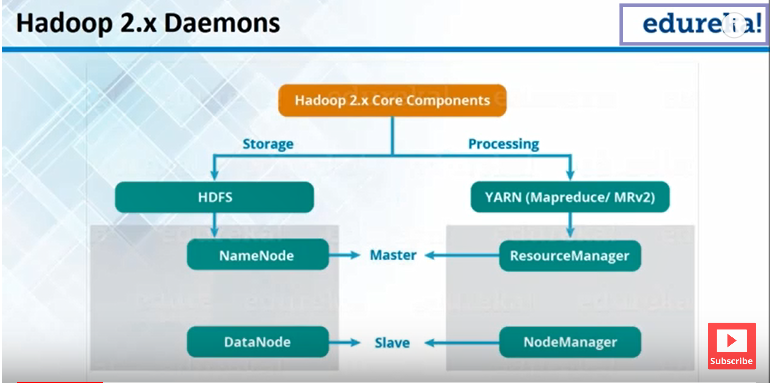


Last but not the least, let us talkabout the **horizontal scaling** or **scaling out** in Hadoop. There are two types of scaling: **vertical** and **horizontal**. In vertical scaling (scale up), you increase the hardware capacity of your system. In other words, you procure more RAM or CPU and add it to your existing system to make it more robust and powerful. But there are challenges associated with vertical scaling or scaling up:

* There is always a limit to which you can increase your hardware capacity. So, you can’t keep on increasing the RAM or CPU of the machine.
* In vertical scaling, you stop your machine first. Then you increase the RAM or CPU to make it a more robust hardware stack. After you have increased your hardware capacity, you restart the machine. This down time when you are stopping your system becomes a challenge.

In case of **horizontal scaling (scale out)**, you add more nodes to existing cluster instead of increasing the hardware capacity of individual machines. And most importantly, you can **add more machines on the go**i.e. Without stopping the system**.** Therefore, while scaling out we don’t have any down time or green zone, nothing of such sort. At the end of the day, you will have more machines working in parallel to meet your requirements.

**Apache HDFS** or **Hadoop Distributed File System** is a block-structured file system where each file is divided into blocks of a pre-determined size. These blocks are stored across a cluster of one or several machines. Apache Hadoop HDFS Architecture follows a Master/Slave Architecture, where a cluster comprises of a single NameNode (Master node) and all the other nodes are DataNodes (Slave nodes).



**Hadoop Daemons:**

Using HDFS Hadoop enable to store big data and using yarn(map reduce ) Hadoop enable process the same big data that we are storing in hdfs

**HDFS has Two main daemons:**

**NameNode :**

Name node is a master daemon that runs on master machine that high-end machine essentially

Name node is responsible for managing data on Hadoop distributed file system.

Name node contains metadata means all data node information, and what data is stored which

data node.

**DataNode:**

data node is a slave machine that runs on commodity hardware

**YARN (Map Reduce) has Two main daemons:**

**ResourceManager:**

Name node is a master daemon that runs on master machine that high-end machine essentially

Resource Manager is responsible for executing processing tasks the over data.

**NodeManager:** node manager is a slave machine that runs on slave machine.

**Note:** We have always one name node and multiple data nodes that running in slave machine

**Note:** Every slave machine has running two daemons

1 datanode

2 nodeManager

As well as master machine has running two daemons

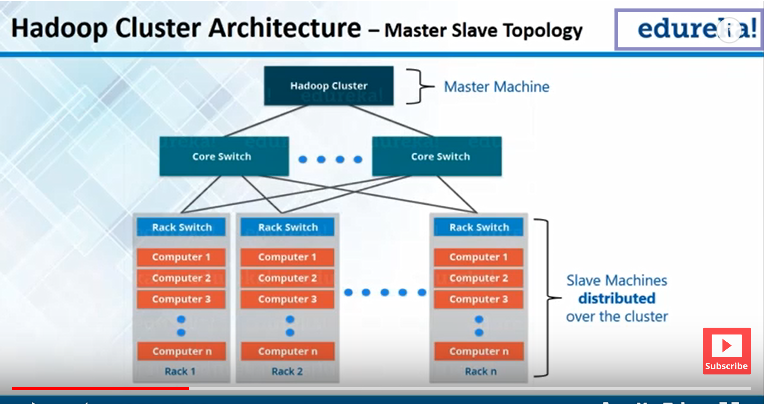
1 NameNode

2 ResourceManager

**Note:** all daemons are processes which run in master and slave machines.

**Heartbeat:** tells name node these data nodes are still available, so name node can use data nodes for stores more date and reads data.

**Hadoop Cluster:**



Hadoop cluster nothing but master slave topology.

Name node, and resource manager are running master machine connected to all slave machines using core switch. All data nodes are stores different racks.

**Rack is group of data node (machines /computers/ slave machines) that present physically at one particular location are connecting to each other.**

**All slave machine connected to master machine using core switch and distributed over cluster and at same time connected to each other**

HDFS Blocks:

Each files are stored on hdfs blocks. The default block size is 128 mb in haddop 2.x 64 mb haddop 1.x

Big data Measurement terms:

1000 Gigabytes(GB) = 1 Terabyte (TB)

1000 Terabytes = 1 petabyte (PB)

1000 petabyte = 1 Exabyte (EB)

1000 Exabyte’s = 1 Zettabyte(ZB)

1000 Zettabyte ‘s = 1 Yottabyte(YB)

**Hadoop**

Hadoop is an open source framework for managing and processing large volume of data.

HDFS is a distributed file system implemented on Hadoop’s framework designed to store vast amount of data on low cost commodity hardware and ensuring high speed process on data.

Hadoop Distributed File System design is based on the design of Google File System. Its notion is “Write Once Read Multiple times".

**Cluster:**

Cluster is group of machines connected throw network and share among them.

Runs on Commodity Hardware: Hadoop can run regular laptop. 8 RM 16 RM

Data locality Optimizations:

Same machine can perform (storage and processing data), means where data is stored we processing, reduces the i/o channels

Hadoop architecture is master /slave architecture

Hadoop 1 = HDFS + MR

Hadoop2 = HDFS + YARN (MRV2)

HDF is responsible for storage, Yarn is a processing Frameworks

1 Master/ n slave 1 master/ N slave

1 Name Node/ N DataNodes 1 Resouce Manager/ N Node Managers

Distributed storage HDFS handle Parallel processing handles YARN

**HDFS Architecture:**

Able to store large amount of data achieving Blocks

Fault Tolerance achieving Replication Factor

Data Locality optimization achieving Distributed Processing

**Limitations:**

As HDFS is designed on the notion of “Write Once, Read multiple times", once a file is written to HDFS, Then it can’t be updated. But delete, append, and read Operations can be performed on HDFS files.

HDFS is not suitable for large number of small sized files but best suits for large sized files. Because file system namespace maintained by Namenode is limited by it’s main memory capacity as namespace is stored in namenode’s main memory and large number of files will result in big fsimage file.

* We can’t update/delete/insert a record into HDFS file, not good idea for store small size files.
* But we can write once, append to it, delete entire file, reading record by record.

**HDFS Important Components**

**Block:**

HDFS is a block structured file system. Each HDFS file is broken into blocks of fixed size usually 128 MB which are stored across various data nodes on the cluster. Each of these blocks is stored as a separate file on local file system on data nodes (Commodity machines on cluster).

HDFS is block structured file system, Each HDFS file is broken into blocks of fixed size usually 128 MB which are stored across various data nodes on the cluster.

Each of these blocks is stored as separate file on local file system on data nodes (commodity machine on cluster).

Any HDFS client trying to access/read a HDFS file, we will get block information from name node first, and then based on the block id’s and locations, data will be read from corresponding data nodes/computer machine on cluster.

**Checking files and blocks in command**

Hadoop fsck / -files –blocks.

**Seeking:**

Moving one control to other control.

By default, and recommends way 128 Mb.

**Replication:**

Replication happens in sequentially.

**Write Operation:**

Write operations happens parallel.

**Name Node:**

* Name node plays a master role in Master/Slave architecture whereas Data nodes acts slaves..
* File System metadata is stored on Name node.
* File System metadata contains majorly, File names, File Permissions and location of each block of files. Thus Metadata is relatively small in size and fits into main memory of computer machine, so it is stored in main memory of name node to allow fast access.
* Name node is single point of contact for accessing files in HDFS and it controls the block ids and locations for data access.

**Important Components of name node:**

**FsImage:** Fs image is a metadata file system it containing File names, File Permission, owners, groups, replication factors, block ids’ and location of each block of files, stored on ram memory.

**Example:**

10 GB => 10 GB/128 blocks

FSimage: FS image file contains about 10 GB file information like below.

Name, File permissions, user, group, replication, block ids, location of blocks, this information called meta data, this record in fs image file (information) takes 4 KB memory.

|  |  |
| --- | --- |
| **Before restart** | **After restart of name node/Hadoop cluster** |
|  | FSimage (records for 100 Files) & Edit Log (0 records) |
|  | 1 file removed and 2 new files got created & 3 records will be inserted into edit log  50 files created & 50 + 3 records are inserted. |
|  | Again Restart Name node  Updated FSimage file (152 records) & edit log is empty(0 records) |
| Only updated fsimage file happens whenever restarted name node. But if edit log file contains million records (Gb storage) in this case 5-6 hours’ time taken for restarting,  To overcome this picture Secondary name node comes in picture.  **Secondary name node:**  Secondary name node every one hour goes to name node and brings the fsimage file present in the name node and edit log (NFS location) and merges two files and updated fs image file, then it will be copy back to primary name node and it flush (edit log empty or 0 records) out edit log for every one hour, so that means edit log will not grow some certain size, even if restart the cluster it will not take more time. Because every one-hour updating fsimage and flushes edit log file.  Note: Secondary name node also maintains the Fsimage copy.  **check point:**  the process of merging fsimage and editlog is called as check point.  **Note:** Secondary name node never act like primary name node means if name node goes down,  It not serves name node requests. only to Reduce the responsibilities of workloads on name node.  **Backup name Node/Journal Node:**  In Hadoop 2.0 architecture brings a concept called high availability, only backup node available in Hadoop 2.0 architecture.  At the time of check point, the name node goes down (or disk issue) so we are losing the fsimage file, overcome this situation always secondary name node updated fsimage file both primary name node and back up name node, so this time backup name node active, why because same fsimage file copy will be contains both primary name node and back up name node.  Whenever restarted in safe mode updated latest edit log and fs image file and server name node requests  **Restart happens some cases:**   * When we have change the configuration * When something goes down. * Hadoop version upgrades * Hadoop software patches installation or bug fixed. | |

**Secondary NameNode:**

Secondary NameNode in hadoop is a specially dedicated node in HDFS cluster whose main function is to take checkpoints of the file system metadata present on namenode. It is not a backup namenode. It just checkpoints namenode’s file system namespace. The Secondary NameNode is a helper to the primary NameNode but not replace for primary namenode.

As the NameNode is the single point of failure in HDFS, if NameNode fails entire HDFS file system is lost. So in order to overcome this, Hadoop implemented Secondary NameNode whose main function is to store a copy of FsImage file and edits log file.

##### **Secondary NameNode Functions:**

1. Stores a copy of FsImage file and edits log.

2. Periodically applies edits log records to FsImage file and refreshes the edits log. And sends this updated FsImage file to NameNode so that NameNode doesn’t need to re-apply the EditLog records during its start up process. Thus Secondary NameNode makes NameNode start up process fast.

3. If NameNode is failed, File System metadata can be recovered from the last saved FsImage on the Secondary NameNode but Secondary NameNode can’t take the primary NameNode’s functionality.

#### **Safe Mode**

Safe Mode in hadoop is a maintenance state of NameNode during which NameNode doesn’t allow any changes to the file system.

During Safe Mode, HDFS cluster is read-only and doesn’t replicate or delete blocks. At the start up of NameNode

* It loads the file system namespace from the last saved fsimage into its main memory and the edits log file.
* Applies/merges edits log file on fsimage and results in new file system namespace.
* And then it receives block reports containing information about block locations from all data nodes

**Data Node:**

* Data nodes are slave machines which are store the actual data of HDFS files.
* Each data node on cluster periodically send a heartbeat message to name node, so for every 10 seconds data nodes are sending message to name node, if name node doesn’t receive heart bit message from particular data node from last 10 minutes (maximum 10 minutes’ name node waiting), then name node mark as dead data node and doesn’t not dispatch any new I/O requests to them
* Data node death may cause the replication factor of some blocks to fall below their specified values.
* Name node constantly tracks which blocks must be re-replicated and initiates replication whenever necessary.

Thus all the blocks on a dead data nodes are re-replicated on other live data nodes and replication factor normal.

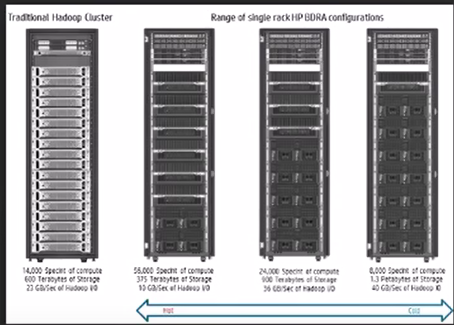
**Block copy algorithm:**

At max 1 copy of a block will be present in any data node.

**Rack Awareness:**

Rack awareness like cabinets, Rack awareness generally contains CPU, RAM, Hard Disk, network channels.

One rack contains max 40 data nodes into 1 rack



At max 2 copy of any block can be stored in one rack, more than two (3) is not possible to store in one rack of replication

factor 3.

The 3rd copy must be stored in other rack.

Awareness of racks, which Racks are closer

Ex: 1 rack present in Jersey and 1 rack present in Hyderabad on cluster, if request from Hyderabad client access the file, name node will give Hyderabad based on distance between client and racks.

Usually Hadoop clusters of more than 30-40 nodes are configured in multiple racks. Communication between two data nodes on the same rack is efficient than the same between two nodes on different racks.

In large clusters of Hadoop, in order to improve network traffic while reading/writing HDFS files, NameNode chooses data nodes which are on the same rack or a nearby rack to read/write request (client node).

NameNode achieves this rack information by maintaining rack ids of each data node. This concept of choosing closer data nodes based on racks information is called Rack Awareness in Hadoop.

A default Hadoop installation assumes all the nodes belong to the same rack.

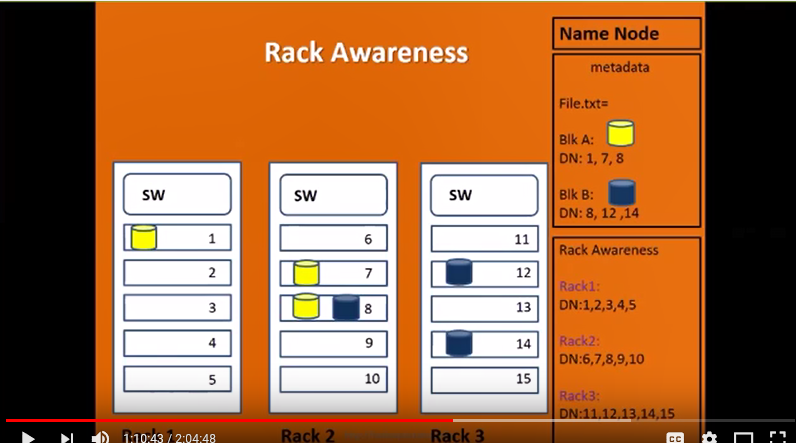
###### **Replica placement via Rack Awareness in Hadoop**

A simple policy is to place replicas across racks. This prevents losing data when an entire rack fails and allows to make use of bandwidth from multiple racks when reading a file.

On Multiple rack cluster, block replications are maintained with a policy that no more than one replica is placed on one node and no more than two replicas are placed in the same rack with a constraint that number of racks used for block replication should be always less than total no of block replicas.

For example,

* When a new block is created, the first replica is placed on the local node, the second one is placed at a different rack, the third one is on a different node at the local rack.
* When re-replicating a block, if the number of existing replicas is one, place the second one on a different rack.
* When the number of existing replicas is two, if the two replicas are on the same rack, place the third one on a different rack;
* For reading, the name node first checks if the client’s computer is located in the cluster. If yes, block locations are returned from the close data nodes to the client.



**Map Reduce:**

Input => splitting => map Phase => shuffle Phase => reducerPhase

**Client submitting job:**

Client => Yarn resource Manager => Node Manager => containers (map/reduce tasks)

Map reduce framework provided two classes:

1. org.apache.hadoop.mapredude.Mpper
2. org.apache.hadoop.mapredude.Reducer

by default, mapper and reducer works. Take the input record and produced result same

in Mapper class contains 4 methods

Mapper {

setup () {

initialization work wants to before processing key value pair

}

map (key k1, value v1, Context context) { } only we override map method in our custom class.

cleanup () {

post processing /closure activities

}

run () {

setup ();

for (every key-value pair) {

map (k1, v1, context)

}  
 cleanup();

}

}

Context: used for writing a text in serializable format on to intermediate out files(storage).

Class WordCountMapper extends Mapper< LongWritable, Text, Text, IntWritable >

{

@override

Public void map (LongWritable, inputkey, Text inputValue, Context context) {

String s1 = inputValue.toString();

String [] words = s1.split(“ “ );

for (String word: words) {

context.write(new Text(word), new IntWritable(1))

}

}

}

Class WordCountReaducer extends Reducer<Text, IntWitable>

{

@override

Public void reduce (Text inputKey, Iterable<IntWritable> values, Context context) {

for (IntWritable i: values) {

sum = sum + i.get();

}

context.write(inputKey, new IntWritable(sum));

}

Public class WordCountDriver {

Configuration conf = new Configuration();

FileSystem fs = FileSystem.gt(count);

Job job = new Job(conf);

Job.setJarByClass(WordCountDriver)

Job.setMapper(WordCountMapper.class)

Job.setReducer(WordCountReaducer.class);

Job.setOutputKeyClass(Text.class)

Job.setOutputValueClass(IntWritable.class)

FileInputFormat.setInputPath(new Path(arg[0]))

FileOutputFormat.setInputPath(new Path(arg[1]))

If (!Job.writeForCompletion(true)) {

return

}

}

Text File => Line offset/number, Line (value)

DataLocality => instead of moving data to where your program, we will move code into data nodes.

|  |  |
| --- | --- |
| **java** | **MapReduce** |
| Int | IntWritable |
| Long | LongWritable |
| Double | DoubleWritable |
| Boolean | BooleanWritable |
| String | Text |

HDFS,

Map reduce- java programing, All file Formats, compression technique, Optimization techniques

Yarn – SQL 95%, java 5%,

pig – SQL 95%, java 5%,

Hive – SQL 95%, java 5%,

Impala - – SQL 95%, c++,

Hbase, -NoSql Databases cassendra, mongodb

Sqoop,

Flume,

Oozie,

Hue

**Real-time Tools:**

Cluodera Manager, Putty, mobaxTeam, Filezilla, winscp, Eclipse, DbVisualizar.

Cloudera Quick Start Vm => 10 % things in real time environment.

**Map Reduce / Mrv1**

In classic mapreduce framework, there are two major components Job Tracker and Task Tracker which work in Master-Slave fashion. The Job Tracker is responsible for allocating resources required to run a mapreduce job and scheduling activities. Task trackers are initiated(task tracker) by Job tracker to process individual tasks. Since the Job tracker is responsible for both resource management (assigning resources to each job) and job scheduling (assigning task to task trackers and monitoring task progress) in a single node, scalability is an issue in large HDFS clusters with more than 4000 nodes

**Job Tracker responsibility alone in Mrv1:**

Allocates the resource (how much of ram and how many core CPUs allocated for given task) required to run a map reduce job and scheduling activities.

Job scheduling:

Application monitoring:

Progress update for client:

So after reaching 4000 nodes limit job tracker not able to respond(scale) (scalability not).

**Yarn (Yet another resource negotiator):**

Yarn is a generic resource management framework; inside resource management framework we can run any processing engine (any distributed framework) like. Map reduce, spark, Tez, that’s reason on the same Hadoop cluster where we have yarn installed, we can able to run both map reduce as well as spark also.

Yarn is replaced by job tracker (Resource manager in yarn) and task tracer(Node Manager in yarn) in Mrv1 map reduce. here Resource manager responsible only just allocates resource across cluster and instructs node manager to allocate resource.

The basic idea of YARN is to split the functionality of resource management into two components

Scheduler

Application Master

**Scheduler:**

it takes care of which job quick started next/run.

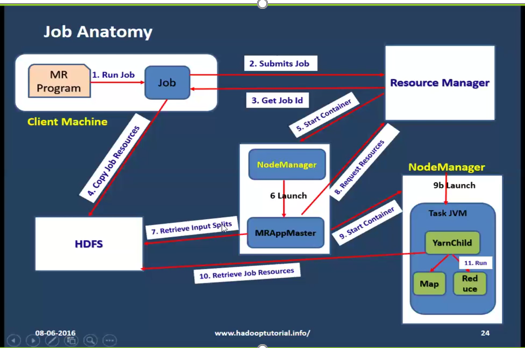
YARN scheduler is responsible for scheduling resources to user applications based on a defined scheduling policy. YARN provides three scheduling options - FIFO scheduler, Capacity scheduler and Fair scheduler.

**FIFO Scheduler** - FIFO scheduler puts application requests in queue and runs them in the order of submission.

**Capacity Scheduler** - Capacity scheduler has a separate dedicated queue for smaller jobs and starts them as soon as they are submitted.

**Fair Scheduler** - Fair scheduler dynamically balances and allocates resources between all the running jobs.

Fair scheduler dynamically balances and allocates resources between all the running jobs. Just after the first (large) job starts, it is the only job running, so it gets all the resources in the cluster. When the second (small) job starts, it is allocated half of the cluster resources so that each job is using its fair share of resources.



Once Client submit the map reduce job, connected to RM, then resource manager return job id and copy job resources in hdfs directory,

Resource manager will be identified one node manager and it will contain container and first start the MRAppMaster,

**Application Master/App master:**

ApplicationMaster is reads input directory size (means how may block contains) based input splits size it calculated how many map and reduces tasks to run then it will request resources to resource manage, resource manager send the list of node manager. Then app master go to available node manger and start the container, in side task JVM, yarn child quick started inside yarn child have map and reduce tasks.

Application master will be only active only when we are running the map reduce jobs and if we are running 100 map reduce jobs at time then 100 application masters running they can be run any available of node manager, once job is completed the application master will be killed.

Application Master collects this progress information from all tasks and aggregate values are propagated to Client Node or user.

ApplicationMaster negotiating(assigning) resources from the ResourceManager and working with the NodeManager(s) to execute and monitor the tasks.

**YARN child:**

After submitting the application, application master dynamically launch YARN child to do the MapReduce tasks.

Yarn child go to the hdfs and retrieve the job resources then start the map and reduce tasks and reporting progress of the map reduce tasks to app master and app master send progress updates to client.

**Node Manager:**

The NodeManager is the slave process that runs on every node in a cluster. Its job is to create, monitor, and kill containers. It services requests from the ResourceManager and ApplicationMaster to create containers, and it reports on the status of the containers to the ResourceManager.

Node manager contains Container inside container map/reduce task are executed.

Ex: 1 container (4 CPU cores, 12 Gb Ram)

**Container:**

A container is an application-specific process that’s created by a NodeManager on behalf of an ApplicationMaster with a constrained(controlled) set of resources (Memory, CPU, etc.)

container is nothing but virtual looks like combination of CPU cores and Ram memory

**Running daemons:**

Resource Manager

Node Manager

Container.

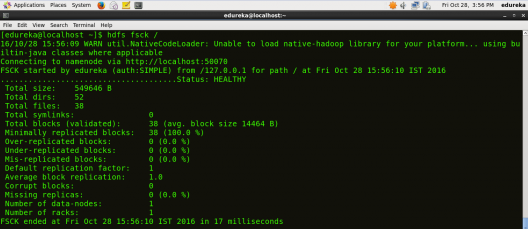
Scheduler

Application Master

## **fsck**

HDFS Command to check the health of the Hadoop file system.

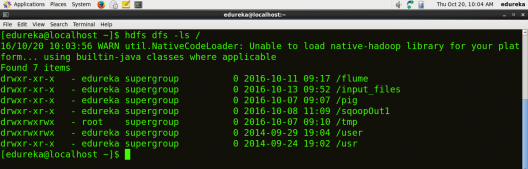
***Command:*** **hdfs fsck /**



## **ls**

HDFS Command to display the list of Files and Directories in HDFS.

**Command:** **hdfs dfs** **–ls /**

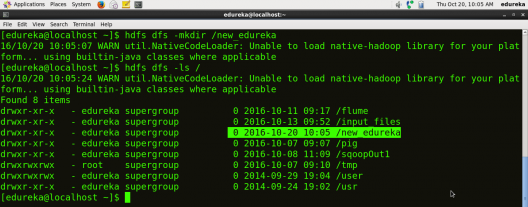


## **mkdir**

HDFS Command to create the directory in HDFS.

***Usage:*** **hdfs dfs –mkdir /directory\_name**

***Command:*** **hdfs dfs –mkdir /new\_edureka**



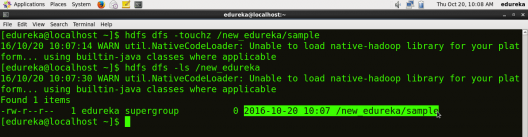
***Note:*** Here we are trying to create a directory named “new\_edureka” in HDFS.

## **touchz**

HDFS Command to create a file in HDFS with file size 0 bytes.

***Usage:*** **hdfs dfs –touchz /directory/filename**

***Command:*** **hdfs dfs –touchz /new\_edureka/sample**



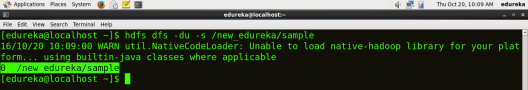
***Note:*** Here we are trying to create a file named “sample” in the directory “new\_edureka” of hdfs with file size 0 bytes.

## **du**

HDFS Command to check the file size.

***Usage:*** **hdfs dfs –du –s /directory/filename**

***Command:*** **hdfs dfs –du –s /new\_edureka/sample**

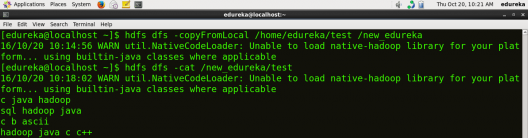


## **cat**

HDFS Command that reads a file on HDFS and prints the content of that file to the standard output.

***Usage:*hdfs dfs –cat /path/to/file\_in\_hdfs**

***Command:*** **hdfs dfs –cat /new\_edureka/test**

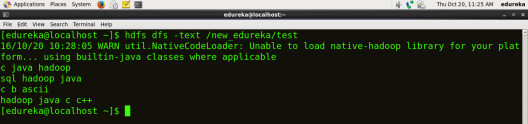


## **text**

HDFS Command that takes a source file and outputs the file in text format.

***Usage:*** **hdfs dfs –text /directory/filename**

***Command:*** **hdfs dfs –text  /new\_edureka/test**

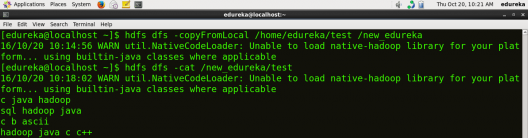


## **copyFromLocal**

HDFS Command to copy the file from a Local file system to HDFS.

***Usage:*** **hdfs dfs -copyFromLocal <localsrc> <hdfs destination>**

***Command:*** **hdfs dfs –copyFromLocal /home/edureka/test /new\_edureka**



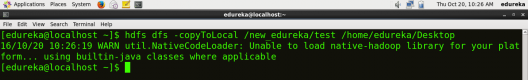
***Note:***Here the test is the file present in the local directory /home/edureka and after the command gets executed the test file will be copied in /new\_edureka directory of HDFS.

## **copyToLocal**

HDFS Command to copy the file from HDFS to Local File System.

***Usage:*** **hdfs dfs -copyToLocal <hdfs source> <localdst>**

***Command:*hdfs dfs –copyToLocal /new\_edureka/test /home/edureka**



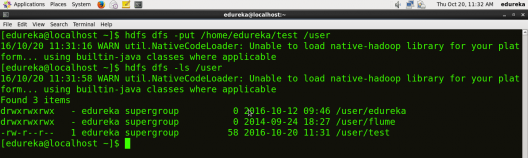
***Note:*** Here test is a file present in the new\_edureka directory of HDFS and after the command gets executed the test file will be copied to local directory /home/edureka

## **put**

HDFS Command to copy single source or multiple sources from local file system to the destination file system.

***Usage:*hdfs dfs -put <localsrc> <destination>**

***Command:*** **hdfs dfs –put /home/edureka/test /user**



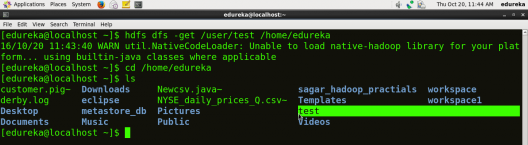
***Note:***  The command copyFromLocal is similar to put command, except that the source is restricted to a local file reference.

## **get**

HDFS Command to copy files from hdfs to the local file system.

***Usage:*** **hdfs dfs -get <src> <localdst>**

***Command:*** **hdfs dfs –get /user/test /home/edureka**



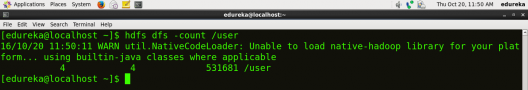
***Note:*** The command copyToLocal is similar to get command, except that the destination is restricted to a local file reference.

## **count**

HDFS Command to count the number of directories, files, and bytes under the paths that match the specified file pattern.

***Usage:*hdfs dfs -count <path>**

***Command:*** **hdfs dfs –count /user**

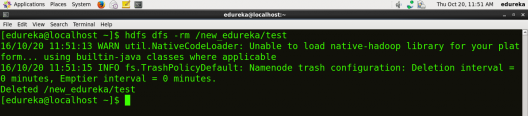


## **rm**

HDFS Command to remove the file from HDFS.

***Usage:*** **hdfs dfs –rm <path>**

***Command:***  **hdfs dfs –rm /new\_edureka/test**

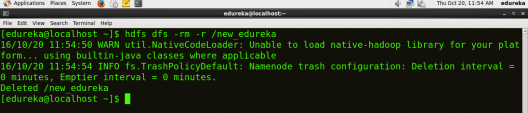


## **rm -r**

HDFS Command to remove the entire directory and all of its content from HDFS.

***Usage:*hdfs dfs -rm -r <path>**

***Command:*** **hdfs dfs -rm -r  /new\_edureka**



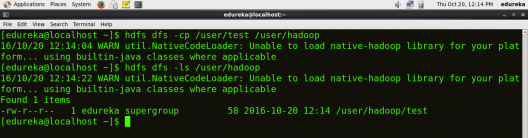
## **cp**

HDFS Command to copy files from source to destination. This command allows multiple sources as well, in which case the destination must be a directory.

***Usage:*** **hdfs dfs -cp <src> <dest>**

***Command:*** **hdfs dfs -cp /user/hadoop/file1 /user/hadoop/file2**

***Command:*** **hdfs dfs -cp /user/hadoop/file1 /user/hadoop/file2 /user/hadoop/dir**

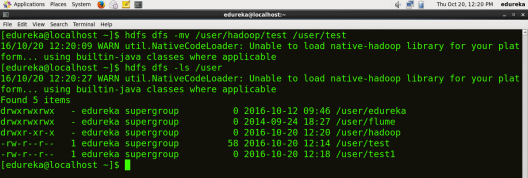


## **mv**

HDFS Command to move files from source to destination. This command allows multiple sources as well, in which case the destination needs to be a directory.

***Usage:***  **hdfs dfs -mv <src> <dest>**

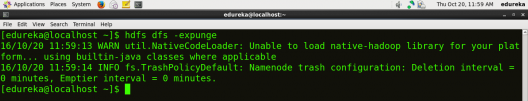
***Command:*** **hdfs dfs -mv /user/hadoop/file1 /user/hadoop/file2**



## **expunge**

HDFS Command that makes the trash empty.

**Command:** **hdfs dfs -expunge**

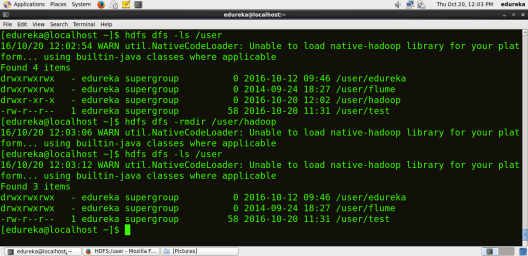


## **rmdir**

HDFS Command to remove the directory.

***Usage:*** **hdfs dfs -rmdir <path>**

***Command:*** **hdfs dfs –rmdir /user/hadoop**

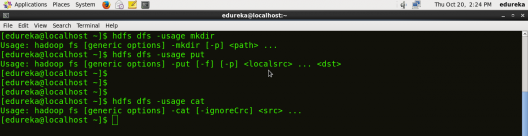


## **usage**

HDFS Command that returns the help for an individual command.

***Usage:*** **hdfs dfs -usage <command>**

***Command:*** **hdfs dfs -usage mkdir**

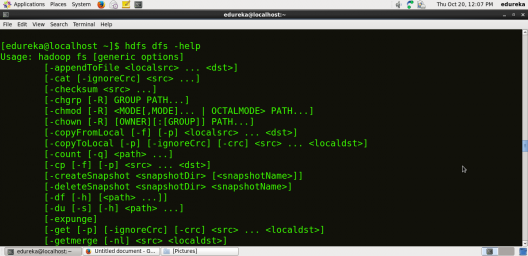


***Note:*** By using usage command you can get information about any command.

## **help**

HDFS Command that displays help for given command or all commands if none is specified.

***Command:*** **hdfs dfs -help**



**MapReduce:**

MapReduce is a programming framework that allows us to perform distributed and parallel processing on large data sets in a distributed environment.

* MapReduce consists of two distinct tasks – Map and Reduce.

**Map Reduce Overview:**

Mapreduce is a framework for processing big data in two phases Map & Reduce. Both the phases have key-value pairs as input and output.

Map phase implements Mapper function, in which user-provided code will be executed on each key-value pair (k1, v1) read from the input files. The output of the mapper function would be zero or more key-value pairs (k2, v2) which are called intermediate pairs. Here the key is what the data will be grouped on and the value is the information related to the analysis in the reducer.

Reduce phase takes mapper output (grouped key-value data) (k2, v2) and runs reduce function on each key-value group. reduce function iterates over the list of values associated with a key and produces outputs like aggregations, statistics etc.. Once the reduce function is done, it sends zero or more key-value pairs (k3, v3) to the final the output file.

By default, Mapreduce input and output file formats are text file formats.

##### **Input Split:**

HDFS splits the input files into equal sized chunks or segments based on minimum split size (mapreduce.input.fileinputformat.split.minsize) property

**Shuffle:**

The shuffle and sort phase is done by the framework. Data from all mappers are grouped by the key, split among reducers and sorted by the key. Each reducer obtains all values associated with the same key.

**Map Reduce Steps:**

* Map takes a data in the form of pairs and returns a list of <key, value> pairs. The keys will not be unique in this case.
* Using the output of Map, sort and shuffle are applied by the Hadoop architecture. This sort and shuffle acts on these list of <key, value> pairs and sends out unique keys and a list of values associated with this unique key <key, list(values)>.
* Output of sort and shuffle will be sent to reducer phase. Reducer will perform a defined function on list of values for unique keys and Final output will<key, value> will be stored/displayed.

**Combiner**

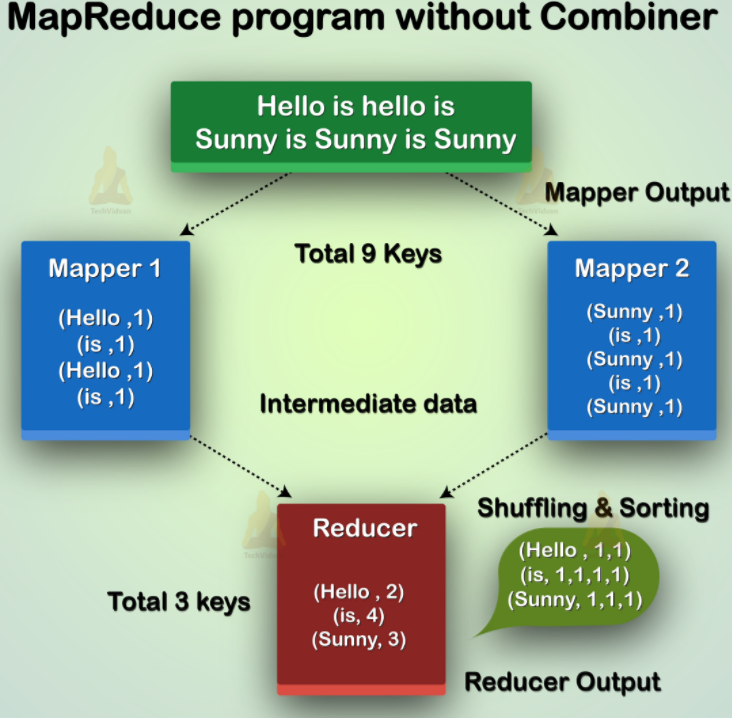
**Combiner** is also known as “**Mini-Reducer**” that summarizes the [**Mapper**](https://techvidvan.com/tutorials/hadoop-mapper-class-mapreduce/) output record with the same Key before passing to the [**Reducer**.](https://techvidvan.com/tutorials/hadoop-reducer/)

**Purpose**

In Mapreduce framework, usually the output from the map tasks is large and data transfer between map and reduce tasks will be high. Since the data transfer across the network is expensive and to limit the volume of data transfer between map and reduce tasks.

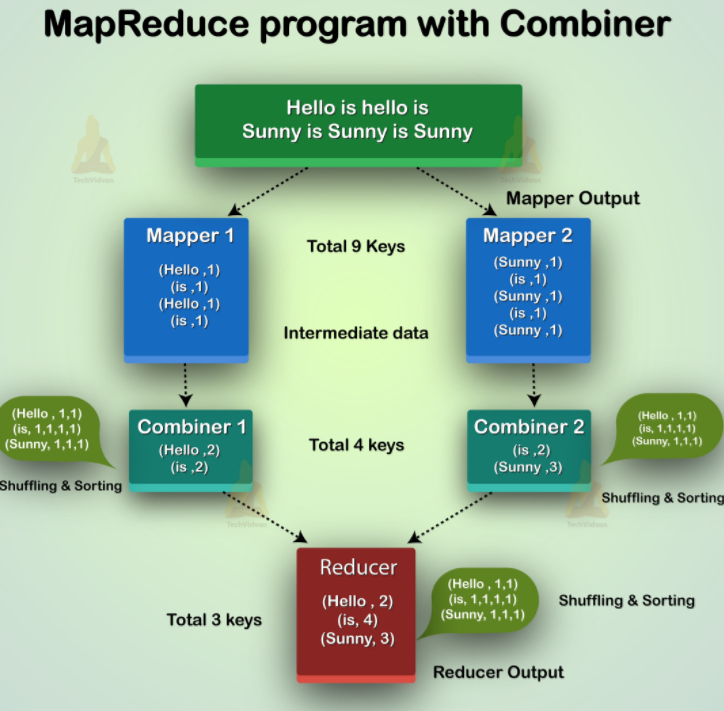
Combiner functions summarize the map output records with the same key and output of combiner will be sent over network to actual reduce task as input.

Example:



As we see in above diagram no combiner is there. Input is split into two mappers. The framework generates 9 keys from the mappers.

So, now we have (9 key/value) intermediate data. Further mapper sends this [key-value](https://techvidvan.com/tutorials/hadoop-mapreduce-key-value-pair/) directly to the reducer. While sending data to the reducer, it consumes some network bandwidth. It takes more time to transfer data to reducer if the size of data is big.



Now from the above diagram, if we use a combiner in between mapper and reducer. Then combiner will shuffle 9 key/value before sending it to the reducer. And then generates 4 key/value pair as an output.

Now, Reducer needs to process only 4 key/value pair data which are generated from 2 combiners. Therefore reducer gets executed only 4 times to produce the final output. Thus, this increases the overall performance.

Advantage:

Let’s now discuss the benefits of Hadoop Combiner in MapReduce.

* Use of combiner reduces the time taken for data transfer between mapper and reducer.
* Combiner improves the overall performance of the reducer.
* It decreases the amount of data that reducer has to process.

Hadoop interview questions:

I am really happy to see all your post and the clarification you provide in a very precise way but i have a question i wouls like to share with you please FYI  
I went for interview last weekend and they asked me for writing a program like some data was given in the below format  
Name age salary  
Him 15 20k  
tim 35 20k  
kim 25 20k  
bim 11 20k  
sim 40 20k  
lim 21 20k  
rim 45 20k

So they asked me to output top 10 salaried record for each age group if(age>10 && age=20 && age=30) return 3; but the output file for each age group only should contain top 10 salaried record Please help me with the logic .