**What is Hadoop?**

Apache Hadoop is an open source software framework used to develop data processing applications which are executed in a distributed computing environment.

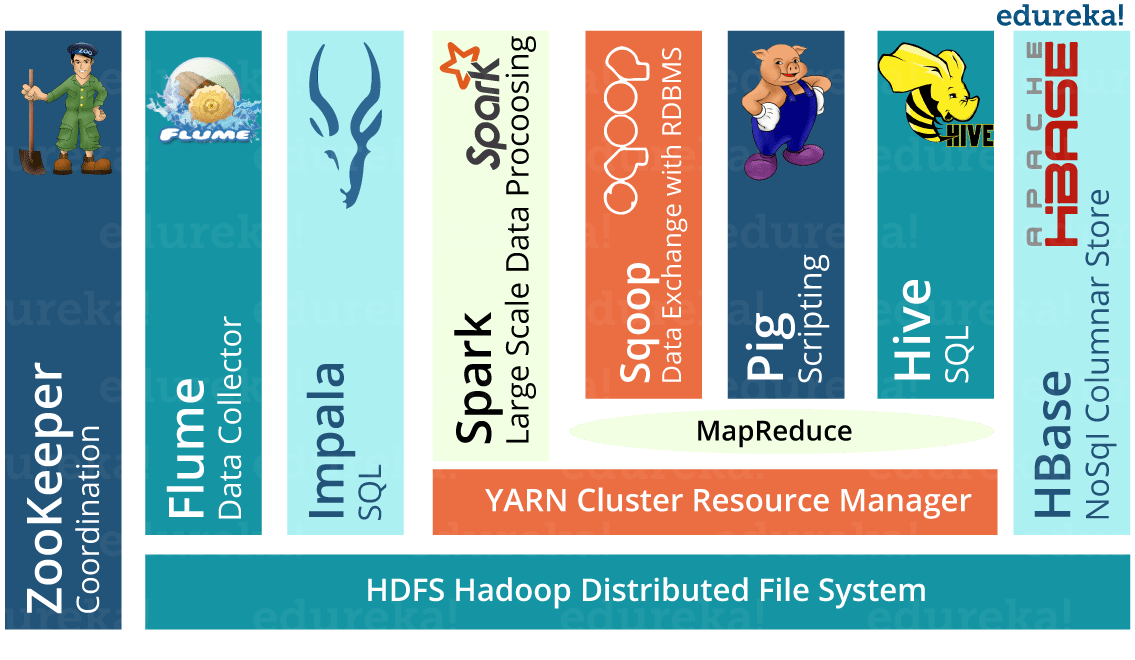
 Applications built using HADOOP are run on large data sets distributed across clusters of commodity computers. Commodity computers are cheap and widely available. These are mainly useful for achieving greater computational power at low cost.

Similar to data residing in a local file system of a personal computer system, in Hadoop, data resides in a distributed file system which is called as a **Hadoop Distributed File system**. The processing model is based on **'Data Locality'** concept wherein computational logic is sent to cluster nodes(server) containing data. This computational logic is nothing, but a compiled version of a program written in a high-level language such as Java. Such a program, processes data stored in Hadoop HDFS.

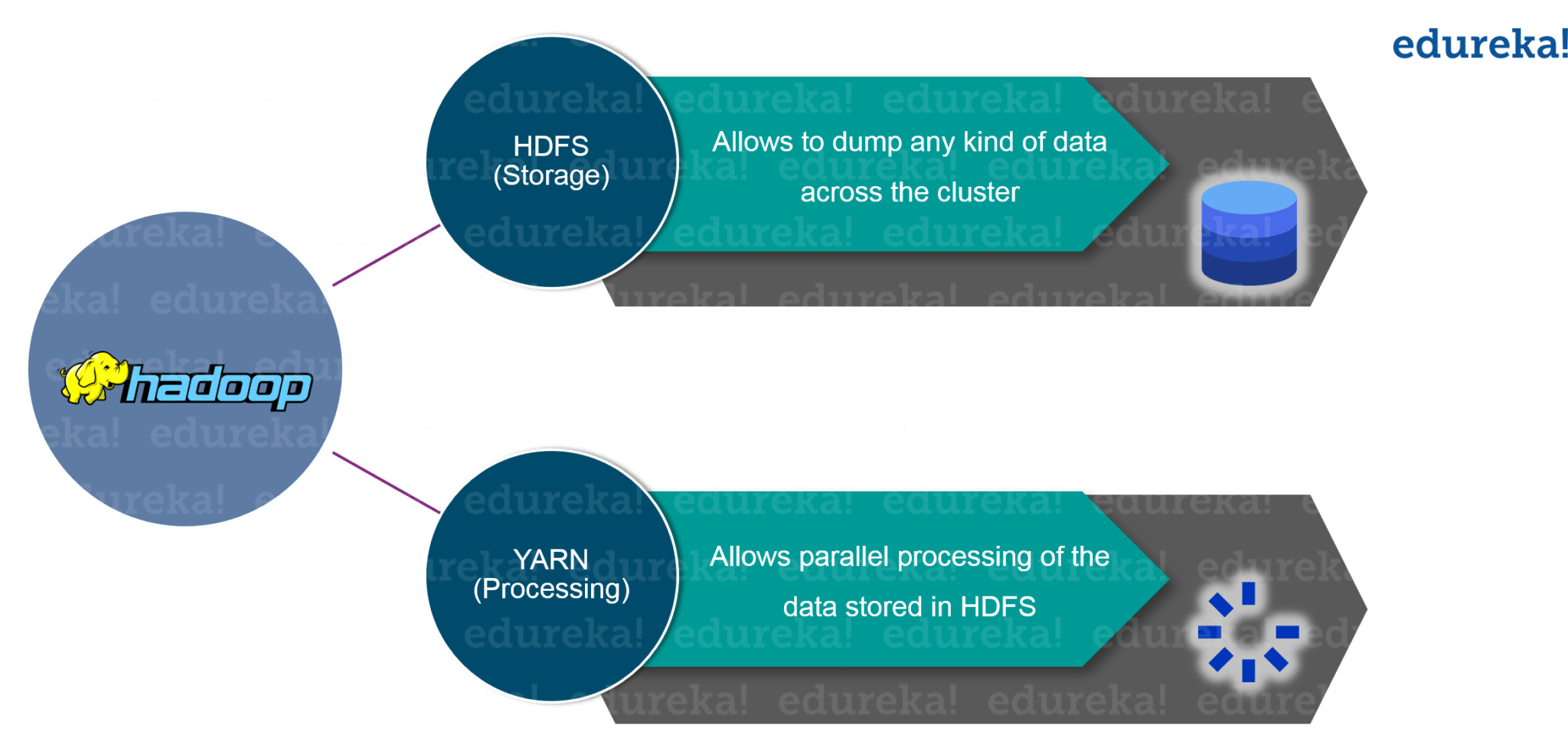
***Do you know?*** Computer cluster consists of a set of multiple processing units (storage disk + processor) which are connected to each other and acts as a single system.

**Hadoop EcoSystem and Components**

Below diagram shows various components in the Hadoop ecosystem-



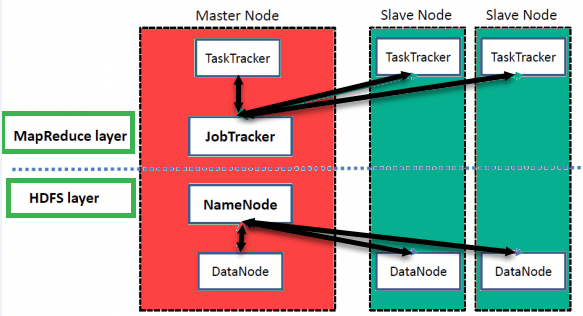
Apache Hadoop consists of two sub-projects –



1. **Hadoop MapReduce:** MapReduce is a computational model and software framework for writing applications which are run on Hadoop. These MapReduce programs are capable of processing enormous data in parallel on large clusters of computation nodes.
2. **HDFS** (**Hadoop Distributed File System**): HDFS takes care of the storage part of Hadoop applications. MapReduce applications consume data from HDFS. HDFS creates multiple replicas of data blocks and distributes them on compute nodes in a cluster. This distribution enables reliable and extremely rapid computations.

Although Hadoop is best known for MapReduce and its distributed file system- HDFS, the term is also used for a family of related projects that fall under the umbrella of distributed computing and large-scale data processing. Other Hadoop-related projects at[Apache](https://www.guru99.com/apache.html)include are **Hive, HBase, Mahout, Sqoop, Flume, and ZooKeeper.**

**Hadoop Architecture**

[](https://www.guru99.com/images/1/hadoop-architecture.png)

High Level Hadoop Architecture

Hadoop has a Master-Slave Architecture for data storage and distributed data processing using MapReduce and HDFS methods.

**NameNode:**

NameNode represented every files and directory which is used in the namespace

**DataNode:**

DataNode helps you to manage the state of an HDFS node and allows you to interacts with the blocks

**MasterNode:**

The master node allows you to conduct parallel processing of data using Hadoop MapReduce.

**Slave node:**

The slave nodes are the additional machines in the Hadoop cluster which allows you to store data to conduct complex calculations. Moreover, all the slave node comes with Task Tracker and a DataNode. This allows you to synchronize the processes with the NameNode and Job Tracker respectively.

In Hadoop, master or slave system can be set up in the cloud or on-premise

**Features Of 'Hadoop'**

**• Suitable for Big Data Analysis**

As Big Data tends to be distributed and unstructured in nature, HADOOP clusters are best suited for analysis of Big Data. Since it is processing logic (not the actual data) that flows to the computing nodes, less network bandwidth is consumed. This concept is called as **data locality concept** which helps increase the efficiency of Hadoop based applications.

**• Scalability**

HADOOP clusters can easily be scaled to any extent by adding additional cluster nodes and thus allows for the growth of Big Data. Also, scaling does not require modifications to application logic.

**• Fault Tolerance**

HADOOP ecosystem has a provision to replicate the input data on to other cluster nodes. That way, in the event of a cluster node failure, data processing can still proceed by using data stored on another cluster node.

**What is HDFS?**

HDFS is a distributed file system for storing very large data files, running on clusters of commodity hardware. It is fault tolerant, scalable, and extremely simple to expand. Hadoop comes bundled with **HDFS**(**Hadoop Distributed File Systems**).

When data exceeds the capacity of storage on a single physical machine, it becomes essential to divide it across a number of separate machines. A file system that manages storage specific operations across a network of machines is called a distributed file system. HDFS is one such software.

**HDFS Architecture**

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.

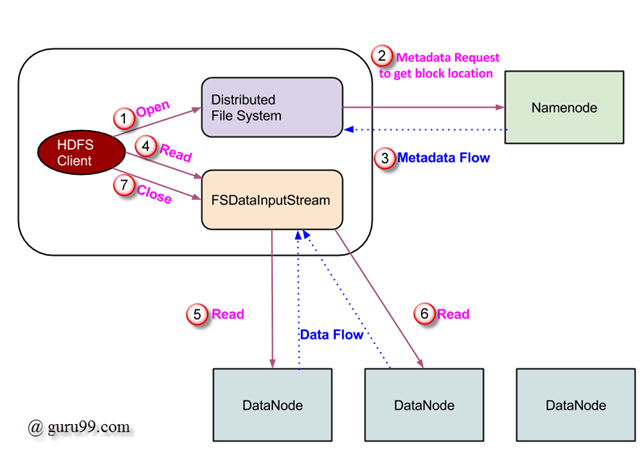
Read/write operations in HDFS operate at a block level. Data files in HDFS are broken into block-sized chunks, which are stored as independent units. Default block-size is 64 MB.

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**

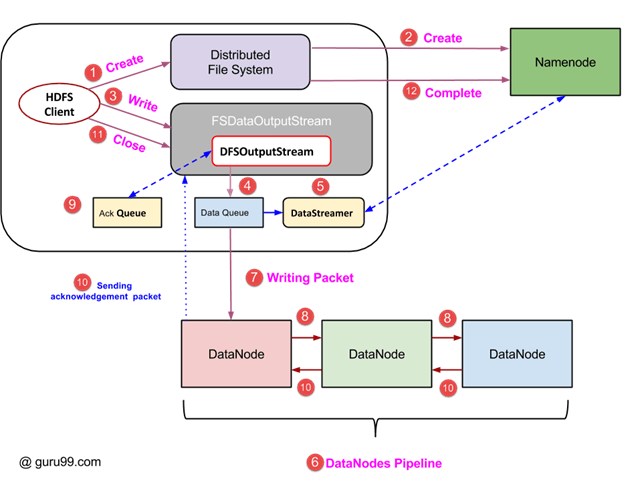
Data read request is served by HDFS, NameNode, and DataNode. Let's call the reader as a 'client'. Below diagram depicts file read operation in Hadoop.

[](https://www.guru99.com/images/Big_Data/061114_0923_LearnHDFSAB1.png)

1. A 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 blocks of a file.
3. In response to this metadata request, addresses of the DataNodes having a 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 the above diagram, a client invokes **'read()'** method which causes **DFSInputStream** to establish a connection with the first DataNode with the first block of a 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 the end of block.
6. Once the end of a block is reached, DFSInputStream closes the connection and moves on to locate the next DataNode for the next block
7. Once a client has done with the reading, it calls **a close()** method.

**Write Operation In HDFS**

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

[](https://www.guru99.com/images/Big_Data/061114_0923_LearnHDFSAB2.png)

1. A client initiates write operation by calling 'create()' method of DistributedFileSystem object which creates a new file - Step no. 1 in the above diagram.
2. DistributedFileSystem object connects to the NameNode using RPC call and initiates new file creation. However, this file creates 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 a client has correct permissions to create a new file. If a file already exists or client does not have sufficient permission to create a new file, then **IOException** is thrown to the client. Otherwise, the operation succeeds and a new record for the file is created by the NameNode.
3. Once a new record in NameNode is created, an object of type FSDataOutputStream is returned to the client. A client uses it to write data into the HDFS. Data write method is invoked (step 3 in the diagram).
4. FSDataOutputStream contains DFSOutputStream object which looks after communication with DataNodes and NameNode. While the client continues writing data, **DFSOutputStream** continues creating packets with this data. These packets are enqueued 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 a 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 a pipeline.
9. Another queue, 'Ack Queue' is maintained by DFSOutputStream to store packets which are waiting for acknowledgment from DataNodes.
10. Once acknowledgment for a packet in the 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 a client is done with the writing data, it calls a close() method (Step 9 in the diagram) Call to close(), results into flushing remaining data packets to the pipeline followed by waiting for acknowledgment.
12. Once a final acknowledgment 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](https://www.guru99.com/java-tutorial.html)interface used for accessing Hadoop's file system.

In order to interact with Hadoop's filesystem 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 command line argument.

**What is MapReduce in Hadoop?**

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.

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

**How MapReduce Works? Complete Process**

The whole process goes through four phases of execution namely, splitting, mapping, shuffling, and reducing.

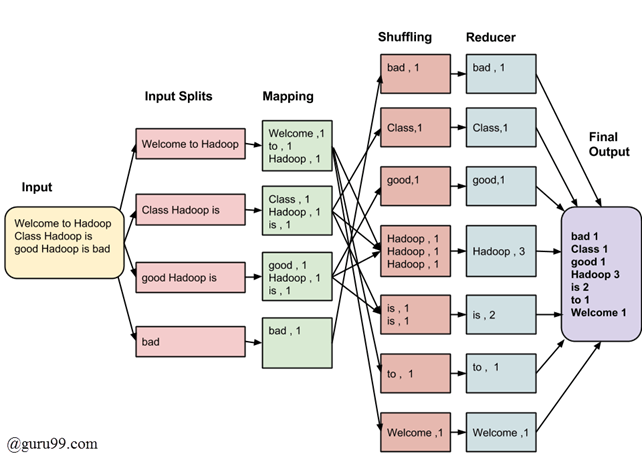
Let's understand this with an example –

Consider you have following input data for your Map Reduce Program

Welcome to Hadoop Class

Hadoop is good

Hadoop is bad

[](https://www.guru99.com/images/Big_Data/061114_0930_Introductio1.png)

MapReduce Architecture

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 the following phases

**Input Splits:**

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

**Reducing**

In this phase, output values from the 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 word.

**MapReduce Architecture explained 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 the 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 to 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 a 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 doesn't work on the concept of data locality. An 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 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:

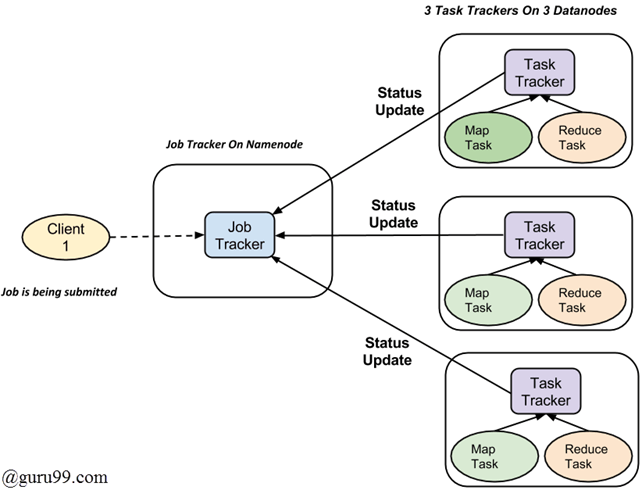
1. **Map tasks** (Splits & Mapping)
2. **Reduce tasks** (Shuffling, Reducing)

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**.

[](https://www.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 job tracker to coordinate the activity by scheduling tasks to run on different data nodes.
* Execution of individual task is then to look after by task tracker, which resides on every data node executing part of the job.
* Task tracker's responsibility is to send the progress report to the job tracker.
* In addition, task tracker periodically sends **'heartbeat'** signal to the Jobtracker so as to notify him of the current state of the system.
* Thus job tracker keeps track of the overall progress of each job. In the event of task failure, the job tracker can reschedule it on a different task tracker.