**Hadoop Architecture**

Hadoop has three main components:

HDFS

YARN  
MAP REDUCE

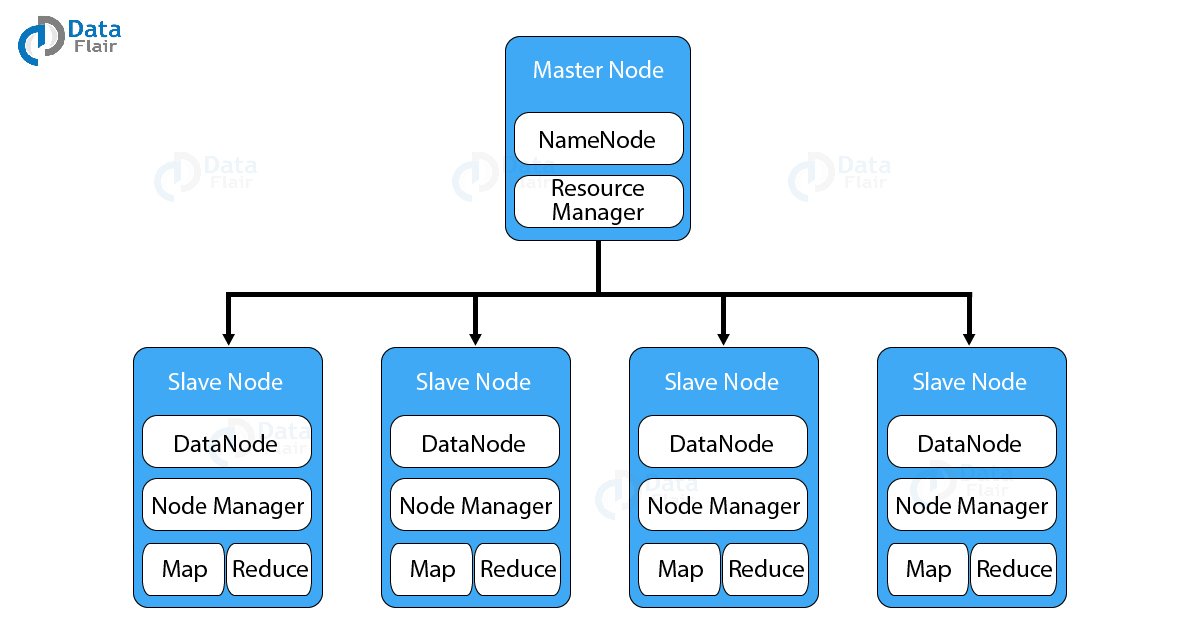
**HDFS**

🡪Hadoop Distributed File System

🡪Storage layer of Hadoop

🡪HDFS splits the data unit into small units called block and stores them in a distributed manner. It has two daemons, one is **Name Node** and other is **Data Node.**

HDFS has a **Master-slave architecture**. The daemon called Name Node runs on the master server. It is responsible for Namespace management and regulates file access by the client. Data Node daemon runs on slave nodes. It is responsible for storing actual business data. Internally, a file gets split into a number of data blocks and stored on a group of slave machines. Name node manages modifications to file system namespace. These are actions like the opening, closing and renaming files or directories. Name node also keeps track of mapping of blocks to Data nodes. This Data nodes serves read/write request from the file system’s client. Data node also creates, deletes and replicates blocks on demand from Name node.



**Java is the native language** of HDFS. Hence one can deploy Data node and Name ode on machines having Java installed. In a typical deployment, there is one dedicated machine running Name node. And all the other nodes in the cluster run Data node. The Name node contains metadata like the location of blocks on the Data nodes. And arbitrates resources among various competing Data nodes.

**Blocks**

The default block size in Hadoop is 64 MB or 128 Mb

Lets understand this with an example:

Suppose u r having a file name abc.txt of size 650 MB.

SO the Name Node will split the data unit into small data unit called block of each size up to 128 Mb(taking 128 MB as default size in this case considering having a huge Hadoop cluster)

So 650 MB will now get split into different block ie

Block1 Block 2 Block 3 Block 4 Block 5 Block 6

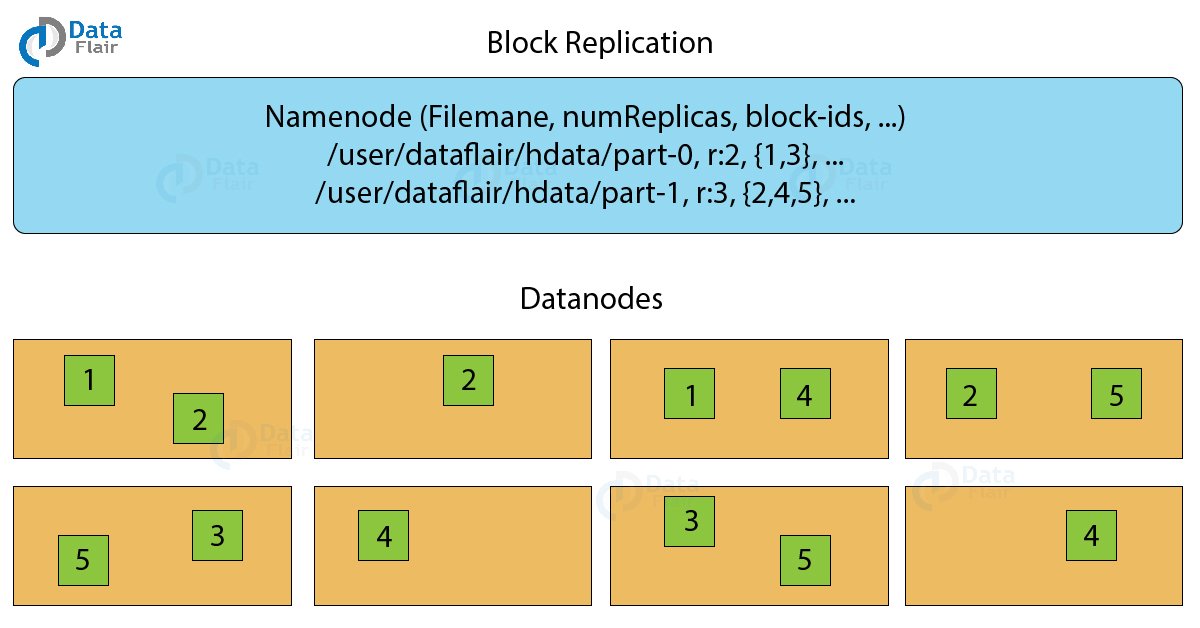
128 Mb 128Mb 128 Mb 128 Mb 128 Mb 10 Mb

What if the block size is 4kb? so for one reason would be data retrieve will me much faster, right?

The reason why we can’t have 4kb block size is because this will create huge metadata which will overload the Name Node. Hence we have to choose our HDFS block size judiciously.

**Replication Management**

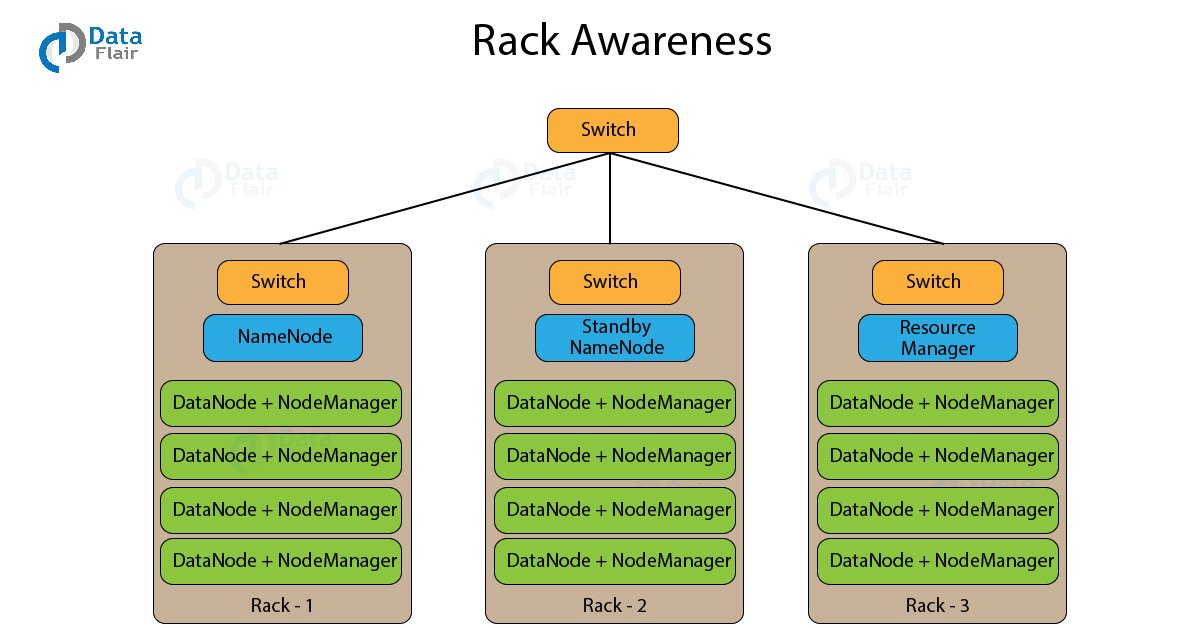
To provide **fault tolerance HDFS**uses a replication technique. In that, it makes copies of the blocks and stores in on different Data nodes. Replication factor decides how many copies of the blocks get stored. It is 3 by default but we can configure to any value.



To maintain the replication factor Name node collects block report from every Data node. Whenever a block is under-replicated or over-replicated the Name node adds or deletes the replicas accordingly.

Rack Awareness

The **Rack** is the collection of around 40-50 Data nodes connected using the same network switch. If the network goes down, the whole rack will be unavailable. A large Hadoop cluster is deployed in multiple racks.



A rack contains many Data node machines and there are several such racks in the production. HDFS follows rack awareness technique to place the replicas of the blocks in a distributed fashion. This rack awareness algorithm provides for low latency and fault tolerance. Suppose the replication factor configured is 3. Now rack awareness algorithm will place the first block on a local rack. It will keep the other two blocks on a different rack. It does not store more than two blocks in the same rack if possible.

**Map Reduce**

[**MapReduce is the data processing layer of Hadoop**](https://data-flair.training/blogs/hadoop-mapreduce-tutorial/). It is a software framework that allows you to write applications for processing a large amount of data. MapReduce runs these applications in parallel on a cluster of low-end machines. It does so in a reliable and fault-tolerant manner.

MapReduce job comprises a number of map tasks and reduces tasks. Each task works on a part of data. This distributes the load across the cluster. The function of Map tasks is to load, parse, transform and filter data. Each reduce task works on the sub-set of output from the map tasks. Reduce task applies grouping and aggregation to this intermediate data from the map tasks.

The input file for the MapReduce job exists on HDFS. The inputformat decides how to split the input file into input splits. Input split is nothing but a byte-oriented view of the chunk of the input file. This input split gets loaded by the map task. The map task runs on the node where the relevant data is present. The data need not move over the network and get processed locally.

Two phase of Map Reduce are

🡪Mapper

🡪Reducer

Mapper task

#### a. RecordReader

The [recordreader](https://data-flair.training/blogs/hadoop-recordreader/)transforms the input split into records. It parses the data into records but does not parse records itself. It provides the data to the mapper function in key-value pairs. Usually, the key is the positional information and value is the data that comprises the record.

#### b. Map

In this phase, the [mapper](https://data-flair.training/blogs/hadoop-mapper-in-mapreduce/) which is the user-defined function processes the key-value pair from the recordreader. It produces zero or multiple intermediate key-value pairs.

The decision of what will be the key-value pair lies on the mapper function. The key is usually the data on which the reducer function does the grouping operation. And value is the data which gets aggregated to get the final result in the reducer function.

#### c. Combiner

The [combiner is actually a localized reducer](https://data-flair.training/blogs/hadoop-combiner-tutoria) which groups the data in the map phase. It is optional. Combiner takes the intermediate data from the mapper and aggregates them. It does so within the small scope of one mapper. In many situations, this decreases the amount of data needed to move over the network. For example, moving (Hello World, 1) three times consumes more network bandwidth than moving (Hello World, 3). Combiner provides extreme performance gain with no drawbacks. The combiner is not guaranteed to execute. Hence it is not of overall algorithm.

#### d. Partitioner

[Partitioner pulls the intermediate key-value pairs](https://data-flair.training/blogs/hadoop-partitioner-tutorial/) from the mapper. It splits them into shards, one shard per reducer. By default, partitioner fetches the hash code of the key. The partitioner performs modulus operation by a number of reducers: key.hashcode()%(number of reducers). This distributes the key space evenly over the reducers. It also ensures that key with the same value but from different mappers end up into the same reducer. The partitioned data gets written on the local file system from each map task. It waits there so that reducer can pull it.

For better understanding how map reduce works watch youtube great learning map reduce

Reduce

#### i. Shuffle and Sort

The reducer starts with shuffle and sort step. This step downloads the data written by partitioner to the machine where reducer is running. This step sorts the individual data pieces into a large data list. The purpose of this sort is to collect the equivalent keys together. The framework does this so that we could iterate over it easily in the reduce task. This phase is not customizable. The framework handles everything automatically. However, the developer has control over how the keys get sorted and grouped through a comparator object.

#### ii. Reduce

The [reducer performs the reduce function](https://data-flair.training/blogs/hadoop-reducer/) once per key grouping. The framework passes the function key and an iterator object containing all the values pertaining to the key.

We can write reducer to filter, aggregate and combine data in a number of different ways. Once the reduce function gets finished it gives zero or more key-value pairs to the outputformat. Like map function, reduce function changes from job to job. As it is the core logic of the solution.

#### iii. OutputFormat

This is the final step. It takes the key-value pair from the reducer and writes it to the file by recordwriter. By default, it separates the key and value by a tab and each record by a newline character. We can customize it to provide richer output format. But none the less final data gets written to HDFS.

