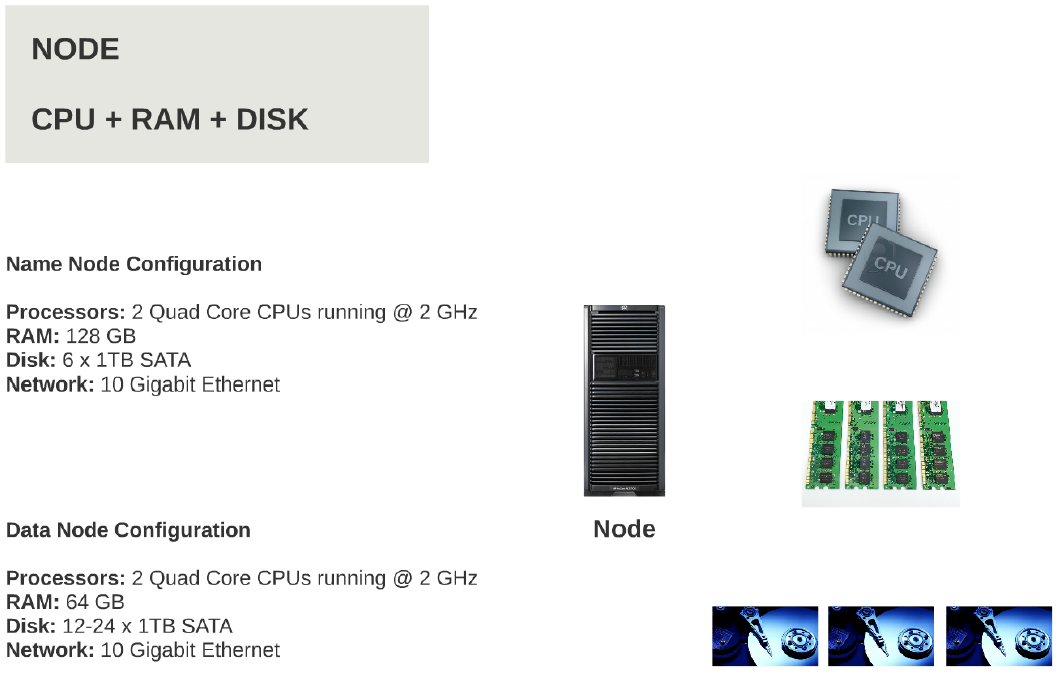
The two main components of Hadoop are -

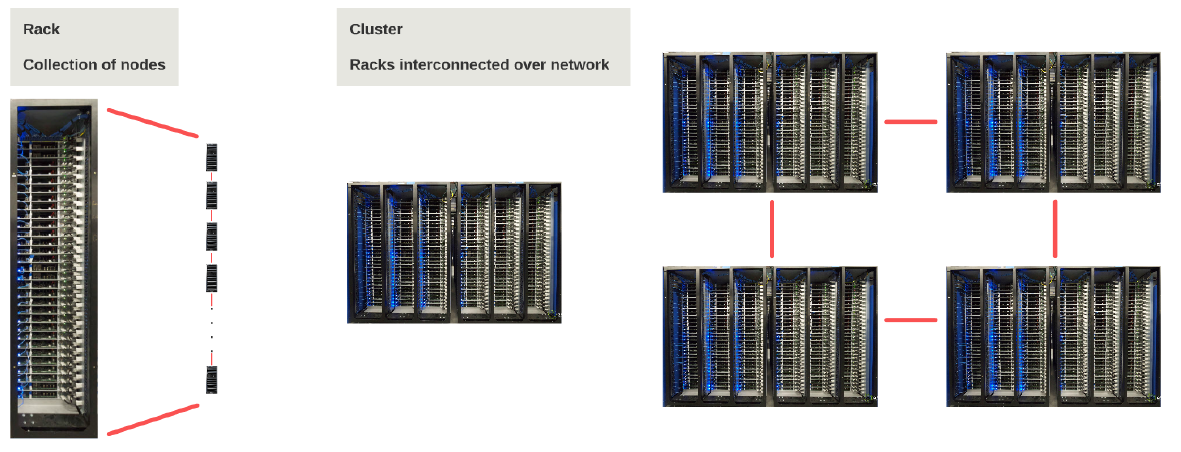
HDFS: In HDFS the data will be stored in distributed fashion across multiple Nodes. HDFS is designed to run on commodity machines which are of low cost hardware. HDFS is highly fault tolerant and provides high throughput access to the applications that require big data.

MapReduce: is a programming model to process the data by Reading/Writing from HDFS.

**Each Node in Hadoop will have its own RAM, CPU and Storage**

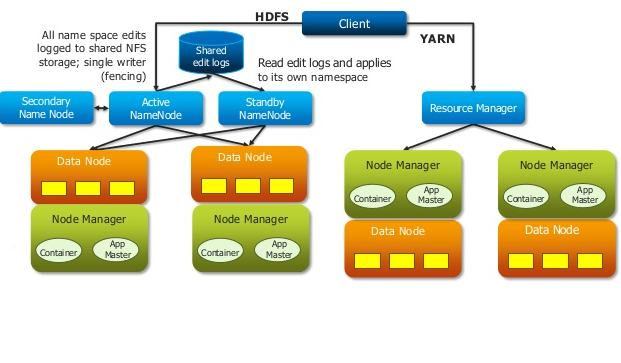


**Node, Rack & Cluster Representation**



**Hadoop Architecture**

Hadoop: is a master/ slave architecture. The master being the namenode and slaves are datanodes. The namenode controls the access to the data by clients. The datanodes manage the storage of data on the nodes that are running on. Hadoop splits the file into one or more blocks and these blocks are stored in the datanodes. Each data block is replicated to 3 different datanodes to provide high availability of the Hadoop system. The block replication factor is configurable.

****

List of Demons or Java Process in MRV1 Architecture:

Hadoop will have the following demons running in the back ground. A Demon is nothing but a java process. All the nodes in the Hadoop clusters will have JVM running on each node as it has to support Hadoop Demons.

* Namenode
* Secondary Namenode
* Job tracker
* Datanode
* Tasktracker

**Hadoop Components**

Name Space: Maintaining the information of files and directories of hierarchical file system is called Name Space.

* When a File/Directory is created it has a name space in its hierarchical order.

Name node: (Metadata of the HDFS)



Namenode is the heart of the Hadoop system. The namenode manages the file system namespace. It stores the metadata information of the data blocks. This metadata is stored permanently on to local disk in the form of namespace image and edit log file. The namenode also knows the location of the data blocks on the data node and when the system starts with this information it knows how to construct a file. If the namenode crashes the entire metadata will be lost and hence hadoop cluster goes down. So it is called single point of Failure (SPOF)

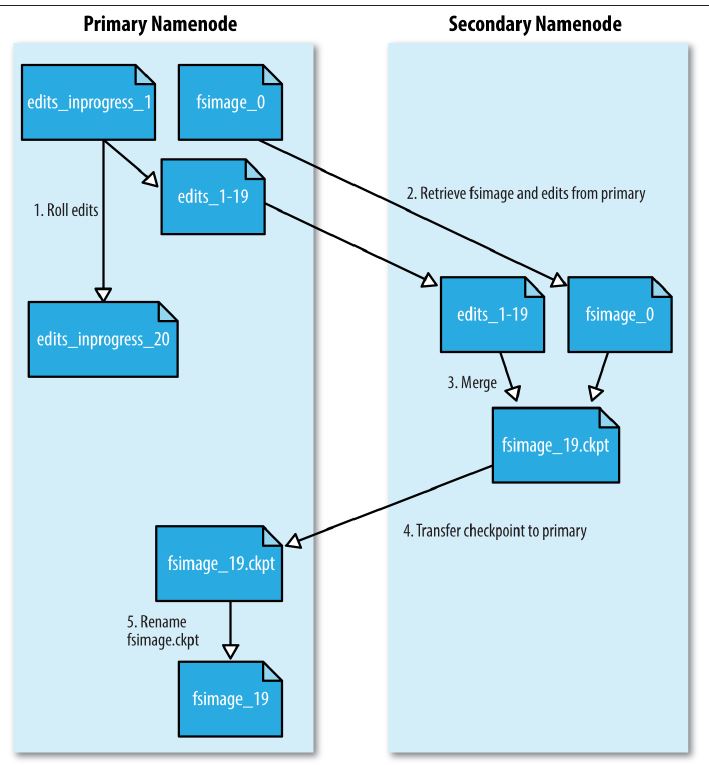
Name node holds the following Meta data information:

1. Replication Factor
2. File Owner
3. File Permissions
4. Created by, Created on, Last modification
5. Block Size, File Size
6. List of block names the file is made up of but not the block locations.

Secondary Name node: (Maintains the Meta data of the Name node)

The responsibility of secondary name node is to periodically (every 1Hr) copy editlog and FSImage by doing FTP from Name Node to Secondary Node and it merges with Secondary name node editlog file and FSimage file. Once it merges with SNN the new files will be copied back to NN. In case if the name node crashes, then the namespace image stored in secondary namenode can be used to restart the namenode.

Note: Name Node and Secondary Name Node are installed on different individual nodes. Both of the Nodes have their own Names Space (editlog and FSImage Files).

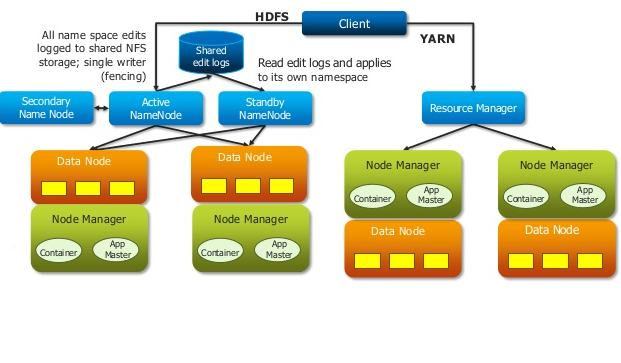


The secondary name node is also called as check point node in later versions of Hadoop as it does check point operations for every 1Hr.

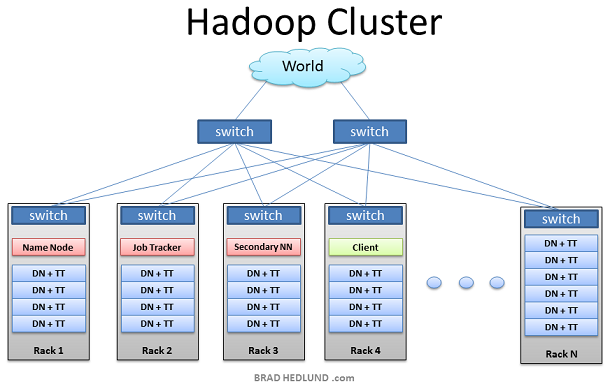
DataNode: (Heart beat & Block Report)

It stores the blocks of data and retrieves them. The datanodes also reports the blocks information to the namenode periodically by sending the heartbeat for every 3 Sec saying the block is alive and block report for every 30 Sec(10th Heart beat).

Below Picture depicts how data nodes are connected in Hadoop cluster environment.



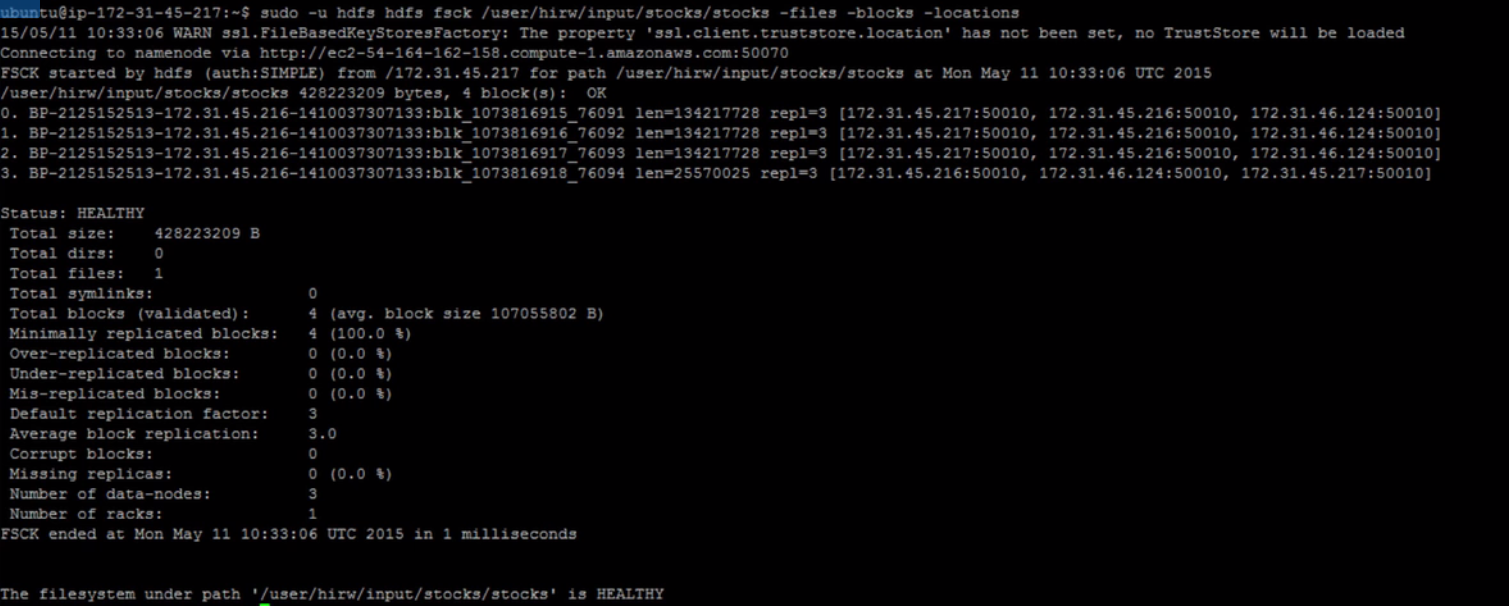
**A typical Hadoop Cluster looks like this:**



Below diagram depicts how data is stored across multiple Nodes:

FSCK is a file system command to check blocks information of a file and their locations across the data nodes.

sudo hdfs fsck /kiran.txt -files -blocks –locations



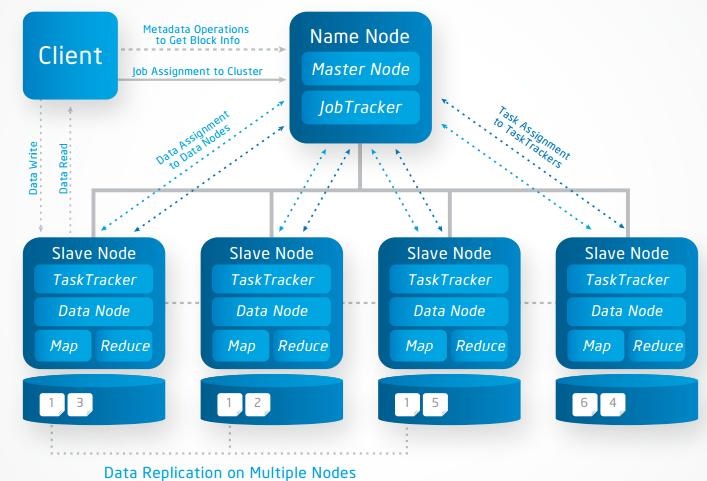
Heartbeat in HDFS:

A heartbeat is a signal indicating that it is alive. A datanode sends heartbeat to Namenode and task tracker will send its heart beat to job tracker. If the Namenode or job tracker does not receive heart beat then they will decide that there is some problem in datanode or task tracker and it is unable to perform the assigned task.

**Job Tracker & Task Tracker in MRV1 Architecture:**

Job Tracker: Job Tracker is a demon which will running on the Namenode. The purpose of Job tracker is to schedule the client jobs by creating map and reduce tasks. This map and reduce tasks will be running as task trackers on the datanodes. Job Tracker also checks for any failed tasks and reschedules the failed tasks on another datanode.

In short we can say Job trackers acts as a Scheduler and Resource Manager.

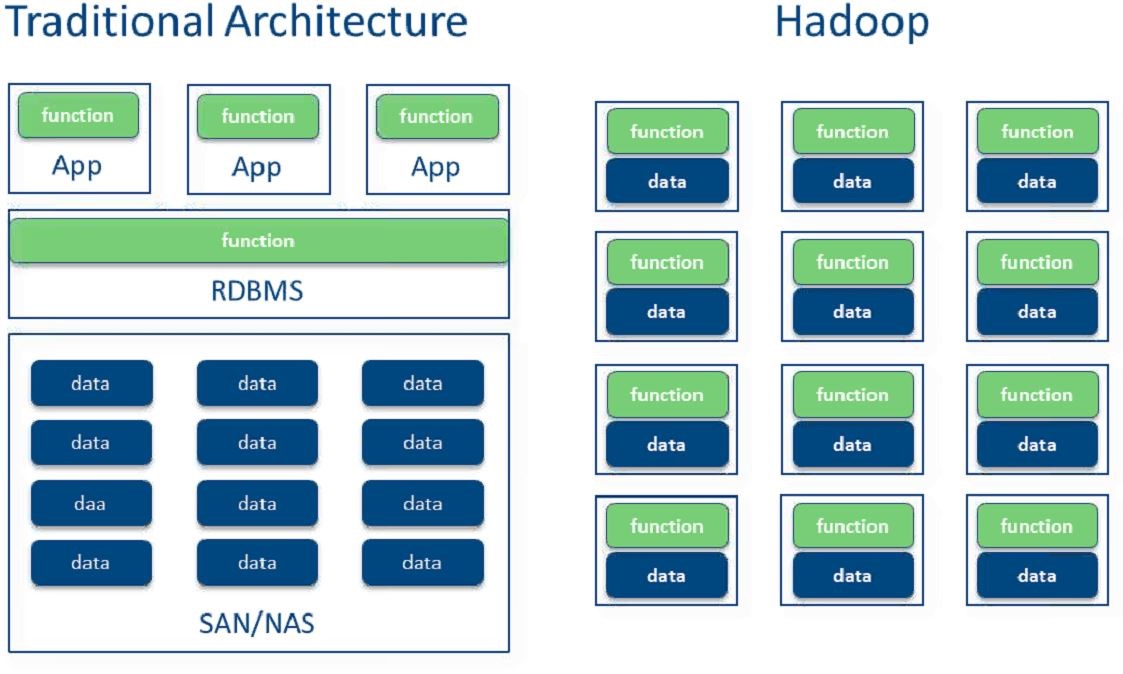


Task Tracker: Tasktracker runs on the datanodes. Task trackers responsibility is to run the map or reduce tasks assigned by the Job tracker. The task tracker will report the status by sending heartbeats to Job tracker periodically.

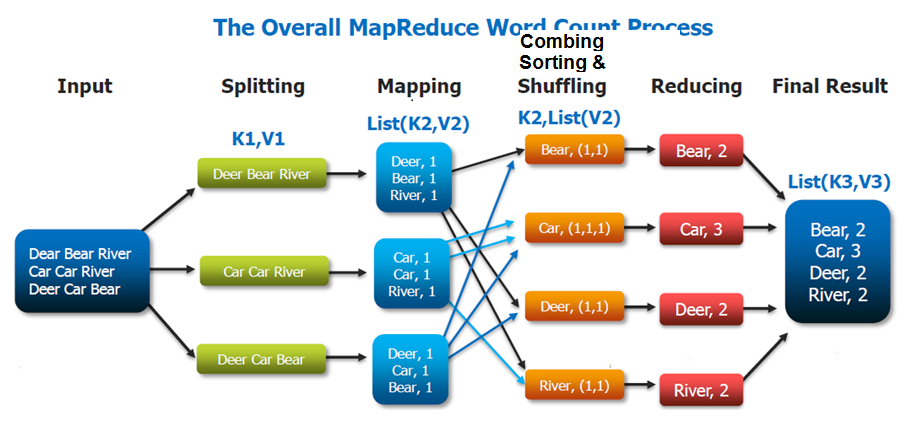
**Map Reduce Logical Flow Diagram**

The reason behind why map reduce is so popular is because of the program is sent where data resides i.e map reduce will be executed on each data node in terms of tasks (mappers and reducers).

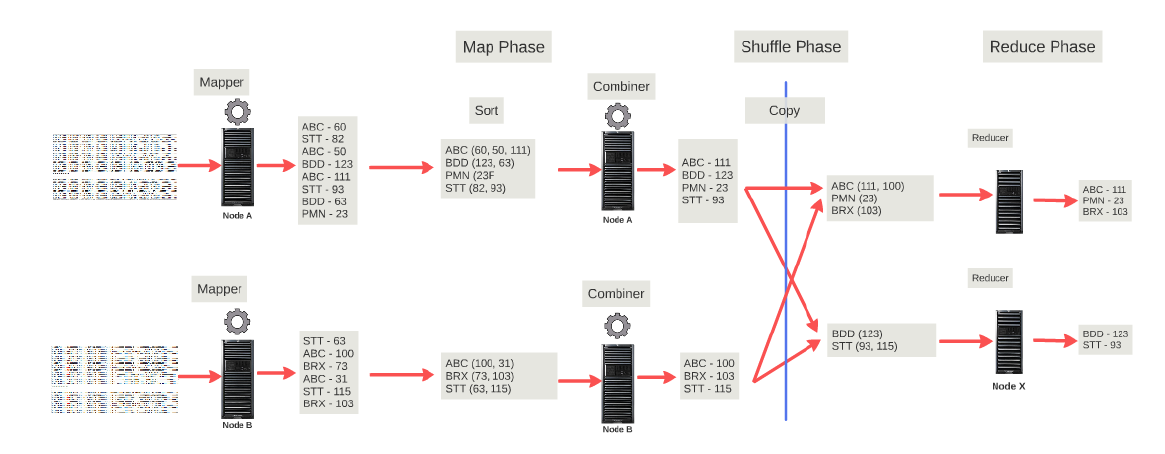
If we see the traditional processing techniques the data is brought where program is running. If the volume of data increases bringing the data to program and writing it back to storage is a performance bottle neck.



Map reduce Flow Chart:



Maximum Closing Price of each symbol for stocks dataset in Mapreduce Program:



Input Split: Each file is logically split as per the end of record and defined block size. Number of input splits is equal to no of mappers and record readers.

10 input splits=10 record readers=10 mappers

Record Reader: Record reader is an interface which will read the file with (Key, value) pair from each input split.

Record Reader can read from 4 types of file-

**Input File Formats:**

1. TextInputFormat
2. KeyValueTextInputFormat
3. SequenceFileInputFormat
4. SequenceFileAsTextInputFormat

If don’t specify format to record reader by default it will take **TextInputFormat**

For a Text file (Byteoffset, EntireLine) will be Key, Value for the Record reader.

Map stage: The mapper will always take (key1, value1) pair as input and process as per the mapper logic and gives (key2, value2) pair as output.

The map function is applied to each record of the input split. This is something similar to row function in Sql which is applied for each row.

Sort & Shuffle: Sorting and Shuffling based on key values of mappers will be done at this stage.

Ex: (hi,4) (hello,3) (how,1)

Reducer Phase: The Reducer phase takes each key-value collection pair from the Combiner phase, processes it, and passes the output as key-value pairs. Note that the Combiner functionality is same as the Reducer hence combiner is called semi-reducer.

By default only 1 reducer for each map reduce program. However this can be configured to run with more reducers.

No of reducers= No of output files of the map reduce program.

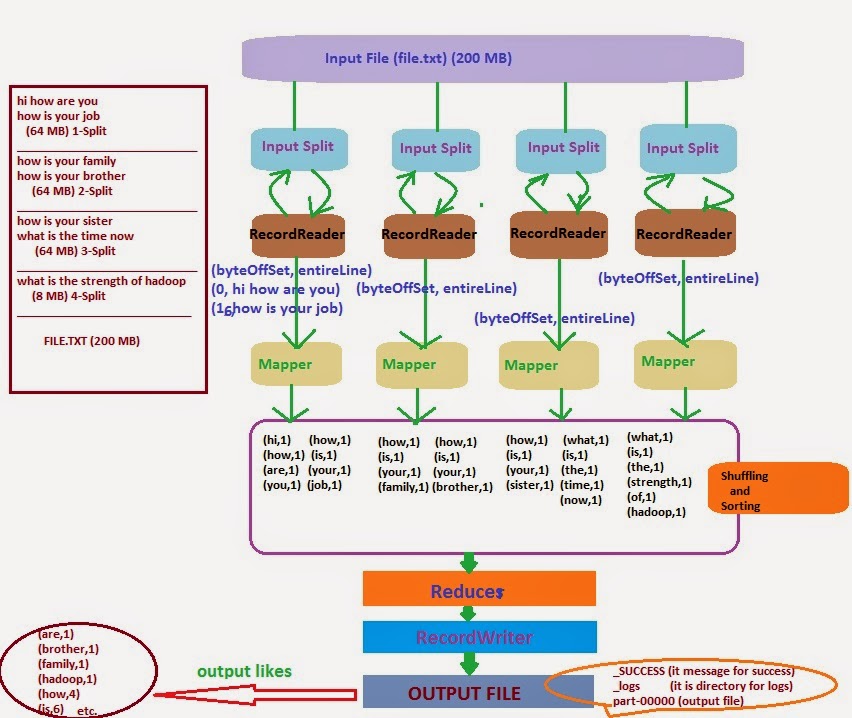
Combine Stage: Combiner phase will come in between sort and shuffle phase. When dealing with large datasets more no of mappers will come into picture to handle those many mappers it will be a performance challenge for a single reducer. Here comes the combiner to shuffle and sort the mappers input. Hence combiner is called mini reducer which is an optional class that operates by accepting the inputs from the Map class and thereafter passing the output key-value pairs to the Reducer class.

The main function of a Combiner is to summarize the map output records with the same key. The output (key-value collection) of the combiner will be sent over the network to the actual Reducer task as input.

No of mappers=No of combiners.

10 input splits=10 record readers=10 mappers=10 combiners.

Record Writer: This is the last phase of MapReduce where the Record Writer writes every key-value pair from the Reducer phase and sends the output as text.



**Read & Writes In HDFS**

How Read & Writes happen in HDFS: When a file is coped to Hadoop cluster from client node or vice versa a java code is executed behind the scenes but not the map reduce code.

Here client can be running on any of the data nodes or it can run on a separate node.

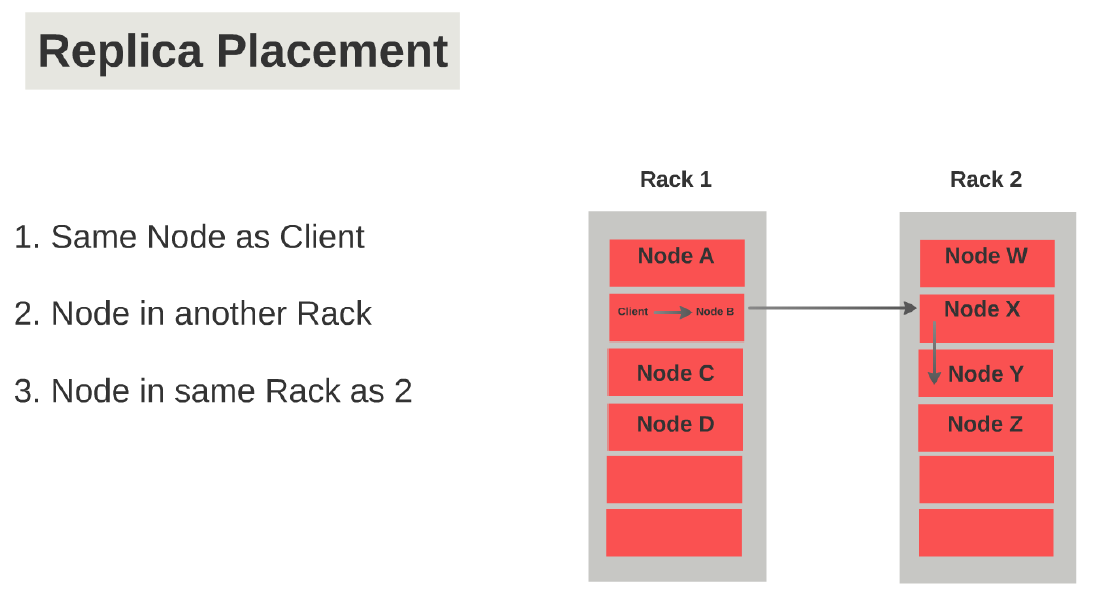
HDFS Replica placement Policy:

* HDFS’s placement policy is to put one replica on one node in the local rack, another on a node in a different (remote) rack, and the last on a different node in the same remote rack.
* If replication factor is more than 3, rest of replicas are placed on random nodes with a constraint that no more than one replica is placed at anyone node and no more than two replicas are placed in the same rack.
* In the below diagram

1st replica 🡪 Rack1 (Node B)

2nd Replica🡪Rack 2 (Node X)

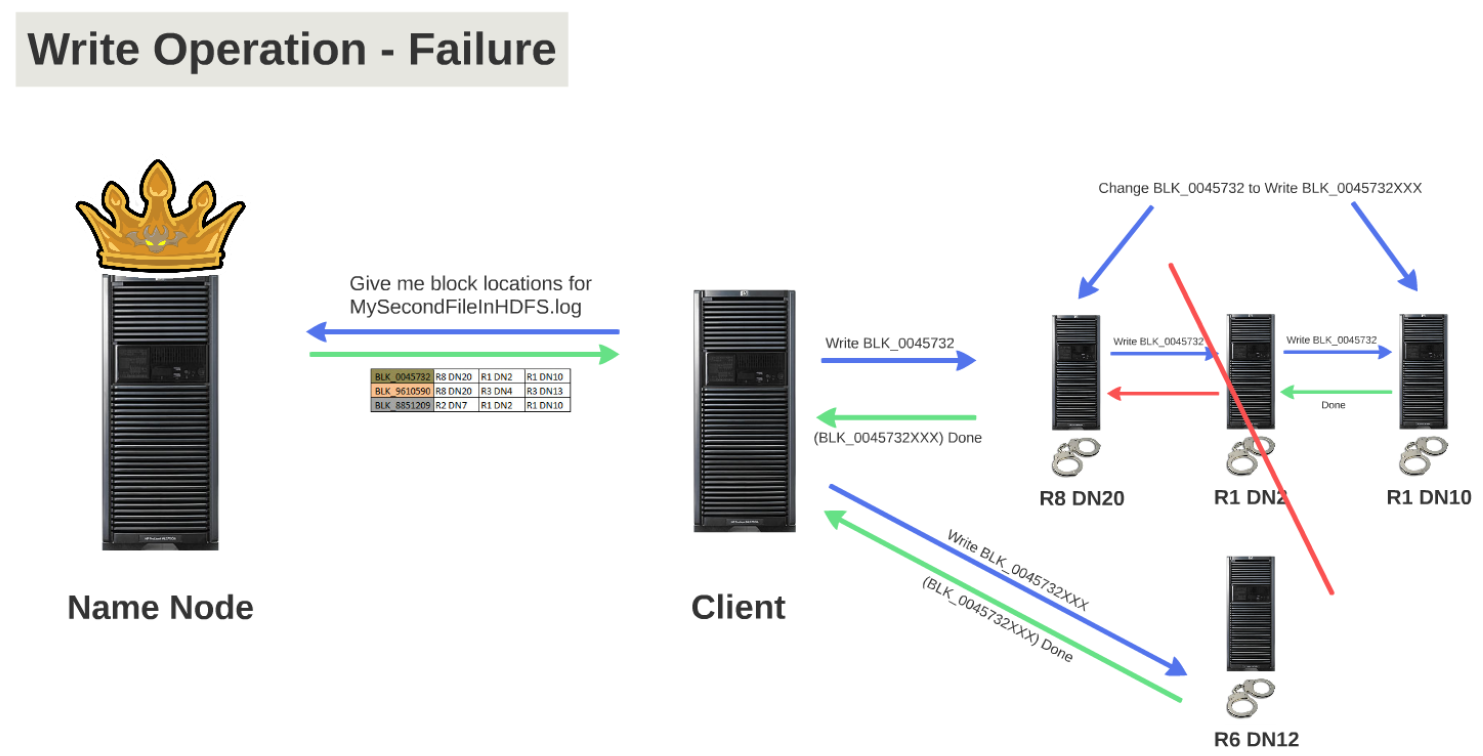
3rd Replica🡪Rack 2 (Node Y)



Write Operations: During write operations it will follow replica strategy. When client want to copy a file to cluster the request goes to Name node and Name node divides the file to blocks and give the block location to client as shown in the below diagram.

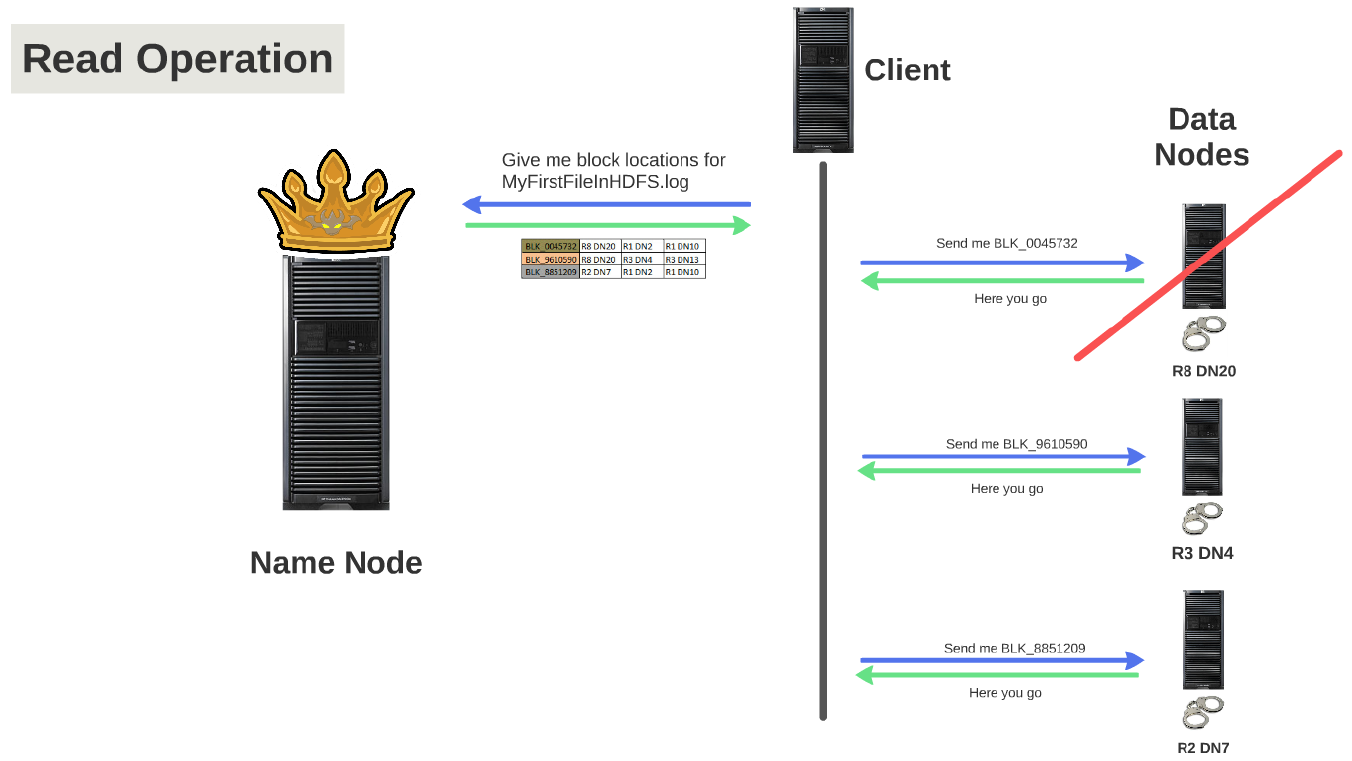
The client will use the block address and copy the blocks of a file in to their respective data nodes. Once the block is copied to data node a check sum calculation is performed to ensure there is no data loss & data corruption in the copied data block. This check sum will be sent as an acknowledgement to Name node.

If let’s say the write request has failed due to data node failure the name node wont attempt to write and the write request will be sent to new data node.



Read operation from HDFS: When a client has requested for block information of a file and having replication factor 3, then Namenode will return the 3 block locations of the file. This block locations will be in the sorted order w.r.t closest location to the client running node.

Then client will start reading from the respective nodes as below. During the read request if data node goes down the read request won’t wait and it will move on to the next available replica node.

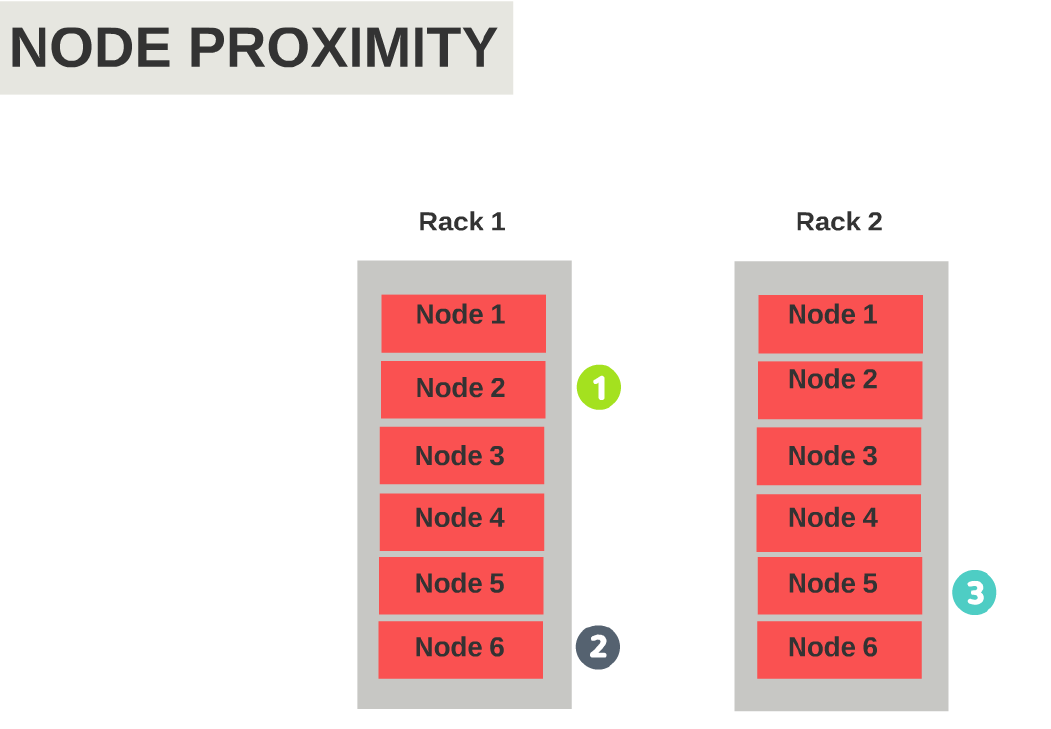


How Nearest Node address calculated for Read Operations:

Case1: If client running on Rack1 Node2, Node2 will be the closest and next closest will be Node6 of Rack1 and next closest will be Rack2 Node5.

Case2: If client is running on some X Node of Rack1 then it will pick the two replicas randomly from the Rack1 and other replica from Rack2.

Case 3: If client is running on R1N1 and if replicas are existing as follows R1N5, R2N2, R2N5 then the closest to client will be R1N5 (First Closest) and Second Closest will be picked randomly from R1N2&R2N5.



Reference for HDFS Read & Write:

<http://www.devinline.com/2015/03/read-and-write-operation-in-hadoop.html>

Hadoop High Availability Architecture (HA): (Active Name Node + (Stand by Node or Passive Node))

The default property of Secondary Name Node to checkpoint periodically from Name Node is 1 Hr.

**dfs.namenode.checkpoint.period**=3600 Sec

If the Namenode goes down in between from its last checkpoint the transactions from its last check point is lost hence Name Node is SPOF (Single Point of Failure).

This kind of failures can be handled in Hadoop HA architecture:

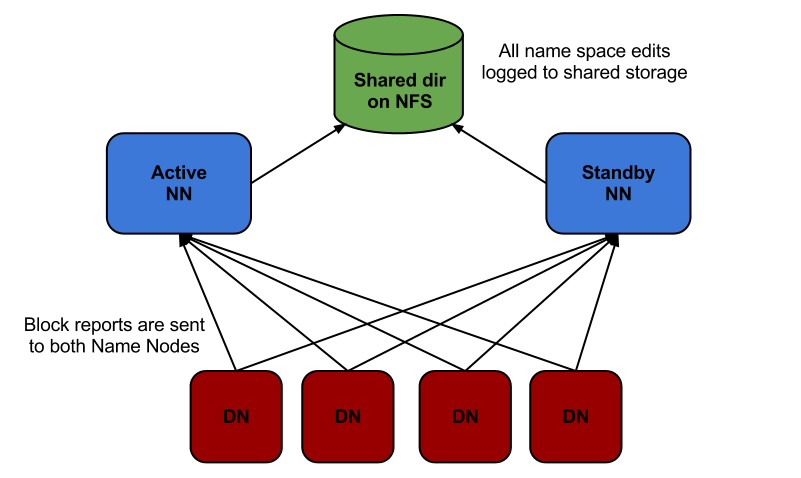
**Shared Storage Using NFS**

In this implementation method, the file system namespace and edit log are maintained on a shared storage device (for example, a Network File System [NFS] mount from a NAS [Network Attached Storage]). Both the active name node and the passive or standby name node have access to this shared storage, but only the active name node can write to it; the standby name node can only read from it, to synchronize its own copy of file system namespace

When the active name node performs any changes to the file system namespace, it persists the changes to the edit log available on the shared storage device; the standby name node constantly applies changes logged by the active name node in the edit log from the shared storage device to its own copy of the file system namespace. When a failover happens, the standby name node ensures that it has fully synchronized its file system namespace from the changes logged in the edit log before it can promote itself to the role of active name node.

Note: At a time only one node can write to editlog in shared drive. This is to prevent corrupting the shared edit log if both the nodes are writing to same file.

Data Nodes will send the block reports to both Active and Standby Nodes.



Reference: <http://www.informit.com/articles/article.aspx?p=2460260&seqNum=6>

HDFS Federation: (Name Node Scalability)

Let say we have 10,000 nodes in our Hadoop cluster, maintaining the name space for all this nodes by a single Name node will be bottle neck and scalability of RAM is not possible after some extent.

HDFS federation made it possible to scale the Name Nodes in a large cluster environment.

In HDFS federation we can scale up the HA configuration. From the below pic each Name Node is associated with Passive Node and each Name node has its own Name Space with individual block pools.

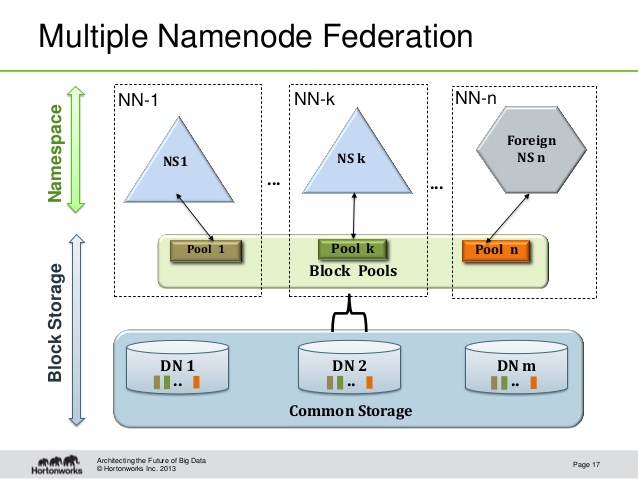
Each name node is responsible for its own block pool and it doesn’t have any coordination with other name nodes. This is something like –

Let’s say we have 2 teams – Marketing and Research in our company funding the Hadoop cluster. You can create a Namespace called /marketing which will be managed by one Namenode and another Namespace under /research which will be managed by another Namenode.

The advantage of this is that you don’t have to run two different Hadoop clusters. You are able to run a single Hadoop infrastructure but one Namenode will manage all the files under /marketing and another Namenode will manage all the files under /research.

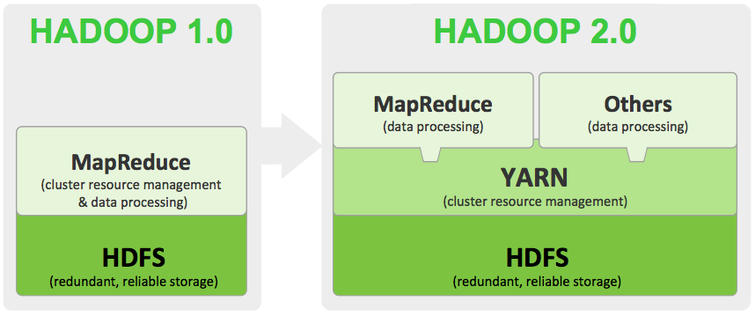
A **Block Pool** is a set of blocks that belong to a single namespace. Datanodes store blocks for all the block pools in the cluster.

From the below diagram if we notice the data blocks have different colors on the data nodes. Each color corresponds one block pool.

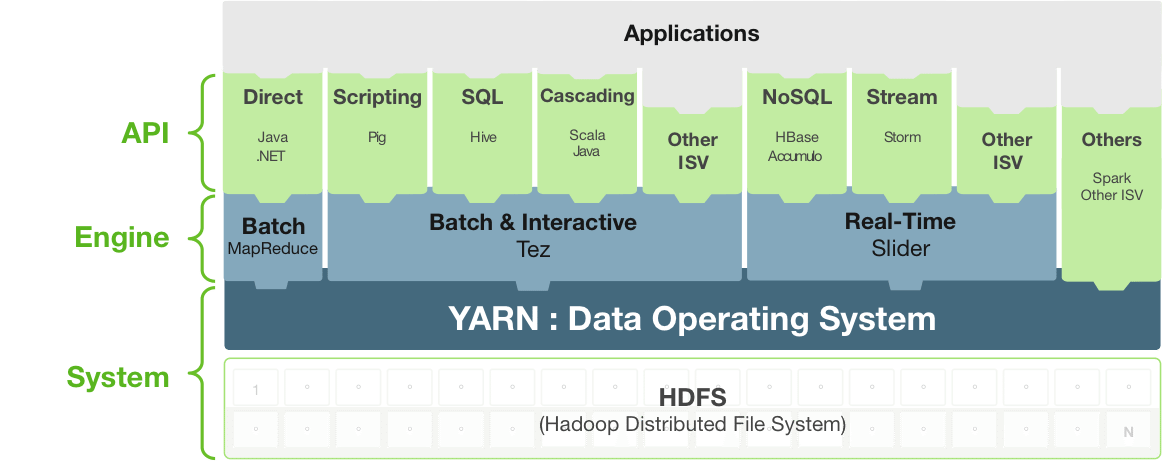


**YARN (Yet another Resource Negotiator)**

Before Hadoop 2.x, Applications use to rely only on MapReduce programming model. After 2.0 version YARN has been introduced to interact with different workloads by running natively in Hadoop.



Apache Hadoop YARN is the data operating system for Hadoop 2, responsible for managing access to Hadoop’s critical resources. YARN enables a user to interact with all data in multiple ways simultaneously, making Hadoop a true multi-use data platform and allowing it to take its place in a modern data architecture. Customers building a data lake expect to operate on the data without moving it to other systems, leveraging the processing resources of the data lake.



YARN Advantages

1. Job Tracker has been divided into two process (**Resource Manager and Application Master**)
2. Nodes can be scalable beyond 4000+.
3. YARN Frame work supports other Distributed Frameworks alongside with MapReduce.
4. Below is the list of Distributed applications which will co-exist with MapReduce in Hadoop which is supported by YARN

* Apache Storm
* Apache Spark
* Apache Drill
* Cloudera Impala
* No-Sql DB’s
* Tez Frame Work

**YARN Architecture**

When we look into MRV1 Architecture –

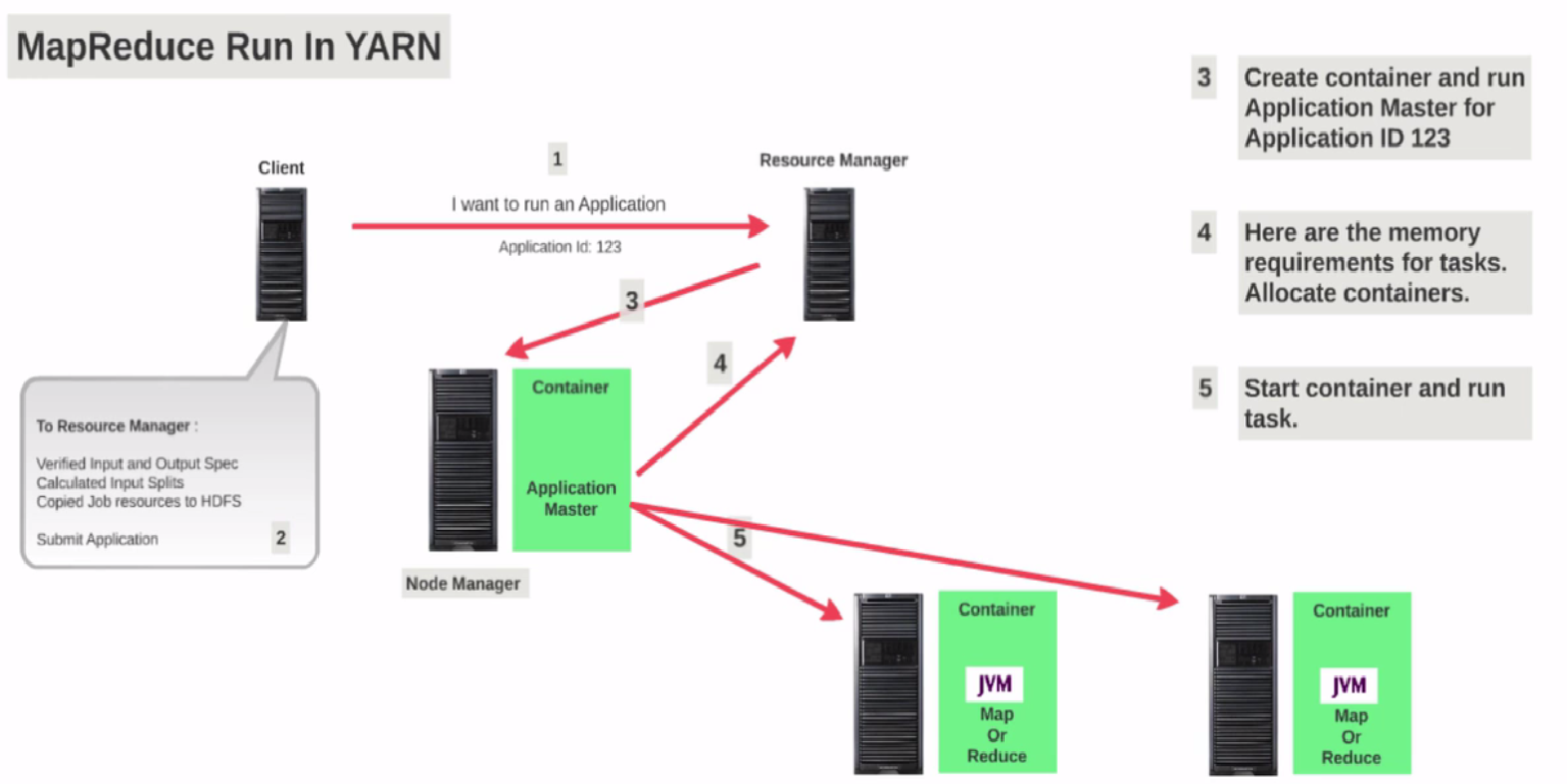
Job tracker takes care of Job coordination, Scheduling (Mapper & Reduce Tasks on DataNode) and Resource Management and Reporting the Job execution process to client

1. Job Tracker is the main demon which handles **(Job Scheduling + Task Monitoring)** so this has become a major performance challenge in MRv1.
2. One more drawback is it supports up to 4000 Nodes only.
3. Available resources cannot be utilized efficiently.

In YARN Architecture

Job Tracker 🡪 (Resource Manager, Application Master)

Task Tracker 🡪 (Node Manager)



Resource Manager: It is the master demon which works in coordination with Name Node and runs on a separate node like Job Tracker.

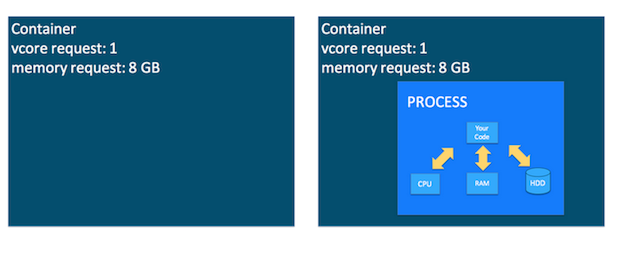
* It is a single process Per-Cluster Level component.
* It keeps on monitoring Applications Master and Node Manager Demons running on the data Nodes.
* Responsible to schedule required resources by allocating containers on data nodes to Applications (that is Per-Application Master)

Application Master: which runs on one of the data node for each application that is been submitted to Resource Manager. Each map reduce job has one application Master.

* Managing assigned Application Life cycle.
* It interacts with both Resource Manager and Node Manager
* It interacts with Resource Manager to acquire required resources.
* It interacts with Node Manager to execute assigned tasks in terms of containers and monitor those task’s status.
* If we submit (10 MapReduce Job 🡪 10 Application) Masters will be created.

Node Manager: is a slave demon which runs on each data node replacing task tracker. When it starts, it announces himself to the Resource Manager. Periodically, it sends a heartbeat to the Resource Manager. Each Node Manager offers some resources to the cluster. Its resource capacity is the amount of memory and the number of vcores.

Containers: Will have set of resources like CPU, RAM & HDD which is been used to execute Map & Reduce Tasks on the Data Nodes.



YARN Flow when client submits Map Reduce Job

When client submits the map reduce job the Resources Manager will identify the no of input splits and corresponding Mappers will be created. Here resource manager will create container for each mapper on the data Nodes along with one Application Master.

Here the Application master will request the resource manager to allocate more resources if the containers doesn’t have enough resources to execute the mappers.

Let say 1 GB of memory is set for Mapper and Reducer with the below property.

* mapreduce.map.memory.mb
* mapreduce.reduce.memory.mb

Referring to above property Resource Manger by default allocate 1 GB of Memory for each container.

Client has submitted a map reduce job of which each Mappers with 2GB in size and container has only 1GB to execute. Here comes Application Master in negotiating the required resources from Resource Manager to get the 2GB of memory for the containers.

Let say 10 Mappers have been created in a job then 10 Containers will be allocated on the Node Managers along with one Application Master. This Application Master also runs on any one the data Node and keeps monitoring the Containers and update the progress of job execution to client.

All the containers of each map reduce job should report the status to Application Master. If a container has failed to execute on a node then application master will request the Resource Manger to allocate the new container on a new node. If Node Manager has failed, the resource Manger which is in constant communication with Node Manager will allocate containers on a new Node Manager.

If Application Master has failed the Resource Manager will allocate on a new Node Manager and the Mappers which are completed will not be re executed. Only the failed/pending mappers will be executed.

Note: Here Application Master is created for each Job and corresponding map/reduce Task execution is handled by its Application Master by reducing the burden on Resource Manager.

Reference: <http://www.journaldev.com/8800/hadoop-architecture-yarn-hdfs-mapreduce>