Chapter 3
Apache Hadoop

Apache Hadoop (Hadoop.apache.org, 2018a) is the most suitable open source ecosystem for processing Big Data in a distributed manner. Google's MapReduce (Dean and Ghemawat, 2008) is the best-proposed programming framework for a Big Data processing solution under the umbrella of Hadoop. Hadoop is not only a framework, but also it is a combination of various tools for storing and processing of Big Data. Hadoop has become more popular due to its adaptability of commodity hardware. Moreover, it has a better edge in terms of performance over a homogeneous environment than a heterogeneous one (Dean and Ghemawat, 2008).

3.1 Hadoop Ecosystem

Hadoop is an open source framework comprising of a set of tools for storage and processing. These tools provide support for executing big data applications. It has a very simple architecture. Hadoop 2.0 version primarily consists of three components: Hadoop Distributed File System (HDFS) (Shvachko et al., 2010), Yet Another Resource Negotiator (YARN) (Vavilapalli et al., 2013) and MapReduce (Dean and Ghemawat, 2008) as shown in fig.3.1:

1. **HDFS:** It allows to split big data among multiple blocks and stores them to various datanodes in the distributed file system. Namenode maintains the metadata for the distributed blocks.

2. **YARN:** It separates the resource management layer and processing components layer. YARN is responsible for managing resources of Hadoop cluster.

3. **MapReduce:** It is a programming framework on top of YARN, responsible for the parallel processing of data that enables enormous scalability across thousand of computing devices run on a Hadoop cluster.

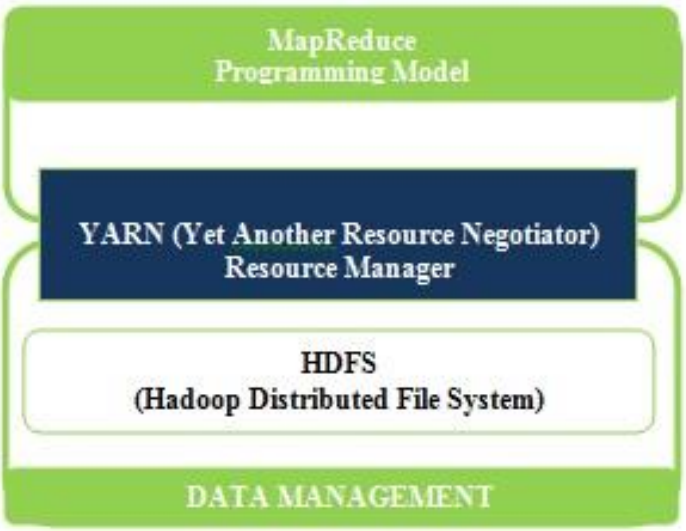

Figure 3.1 Hadoop Architecture

3.1.1 Hadoop Distributed File System (HDFS)

To store data in a distributed environment, Hadoop uses a Hadoop distributed file system. HDFS file system is designed in such a way that it can store and handle very large size of files on commodity hardware. Hadoop does that by dividing large files into small size number of blocks, store them using HDFS block placement policy and access them by providing low-latency data access. Before going to architecture, beneath are the few terms that need to be understood in details:

A. **Very large files:** HDFS allows storing large files which consist of thousands of terabytes, petabytes to zettabytes. These large files split into a number of small size blocks for storage and distribution.

B. **Commodity hardware:** One of the key reasons for the success of HDFS is its adaptability to run on any hardware. It doesn't require a highly configured cluster environment, it can run smoothly even on low-cost computers without any problem.

C. **Blocks:** HDFS splits the large files into number of small blocks for an efficient storage. These blocks are basically 64 MB (default) size, which means that the whole file is divided into 64 MB blocks for reading and writing purpose.

D. **Namenode and Datanodes:** Basically Hadoop cluster nodes are divided into two types of nodes: Two namenodes (primary and secondary) and multiple datanodes. Namenode stores the metadata of file system which basically contains information about datanodes and blocks storage. Datanodes are actual computation devices which perform processing of blocks.

HDFS Architecture:

Hadoop HDFS has master/slave architecture as shown in fig. 3.2. Master node has two components called Resource Manager and Namenode. Slave on each node of a cluster has Node Manager and Datanode. Namenode and Datanode are under the umbrella of HDFS while Resource Manager and Node Manager are under the umbrella of YARN which is discussed in next subsection.

In Hadoop when clients submit applications, it first assigns the job to the master node. The master node then will distribute the task among multiple slaves, to perform computation and the end result will be combined and given back to the master node.

In case of distributed storage, it is important to give indexing for a faster and efficient data access. The namenode that resides on the master node contains the index of data that resides on different datanodes. Whenever an application requires the data, it contacts the namenode that routes the application to the datanode to obtain the data.

Hardware failures are bound to happen, but Hadoop has been developed with an efficient failure detection model. Hadoop has *de-facto* fault tolerance support for data. By default, Hadoop maintains three copies of file blocks on different nodes. Therefore, even in case one datanode fails, the system would not stop running as the data would be available on one or more different nodes.

Fault tolerance does not handle the failure of only slave nodes, but it also takes care of the failure of a master node. Hadoop architecture is fault tolerant against a single point of failure. Hadoop maintains multiple copies of a name node on

different computers as well as maintains two masters, main master aka primary namenode and a backup master aka secondary namenode.

The programmer need not worry about the questions like where the file is located, how to manage failure, how to split computational blocks, how to program for scalability etc. Hadoop HDFS implicitly manages all these efficiently. It is scalable, and its scalability is linear to the processing speed.

In Hadoop 1.x version, managed resources as well as computation. However, Hadoop 2.x splits these two responsibilities into separate entities by introducing YARN.

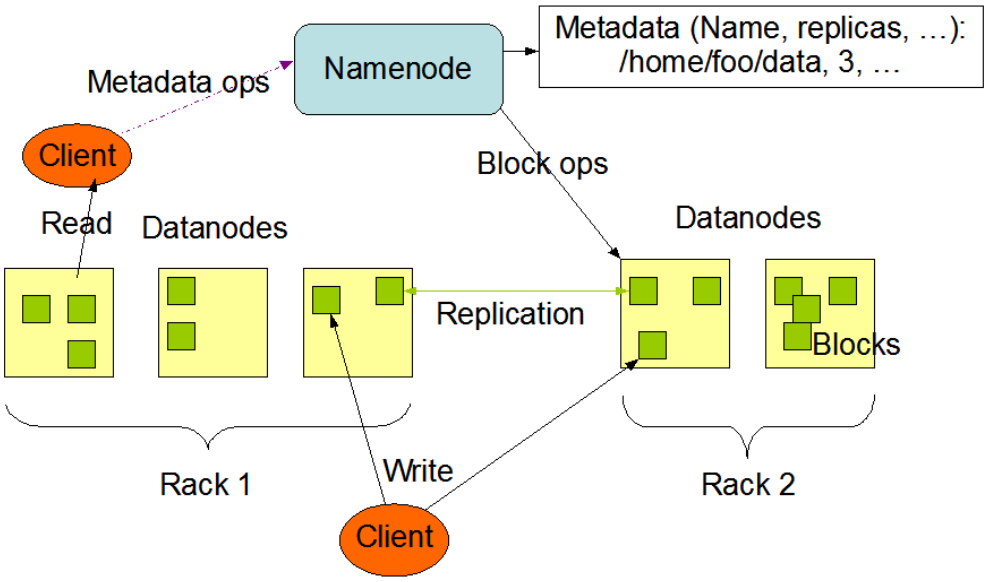

Figure 3.2 HDFS Architecture (Hadoop.apache.org, 2018b)

3.1.2 YARN

YARN is a framework to develop and/or execute applications on distributed resources. YARN supports MapReduce, Tez, and other programming models at the same time play a vital role in enhancing scalability through effective resource utilization. In YARN, Resource Manager (RM), Node Manager (NM), Application Master (AM) and containers are key components.

A. **Resource Manager:** RM is responsible for handling all user requests and schedules the jobs/applications using scheduling management. For each application, RM allocates one Application Master and required resources.

B. **Node Manager:** Each datanode contains a single node manager which is responsible for allocating container for a job and managing individual datanode resources.

C. **Application Master:** For resource allocation of each application, RM negotiates with an Application Master. The Application Master allocates and monitors task containers for each application.

D. **Containers:** Containers are actual processing units for tasks. For processing, containers use resources of nodes, such as a disk, a CPU, memory**,** network, etc. YARN assigns resources (CPU and memory) for the containers of each task using AM. The YARN allocates container resources dynamically as and when required/requested by tasks.

YARN Architecture:

Components in the YARN-based systems are Resource Manager (RM), Application Master (AM) for each application, Node Manager (NM) for each slave node (datanode) and an application container for each application running on a Node Manager. Resource Manager has two main components: a Scheduler and an Application Manager. The scheduler schedules the tasks based on the availability and requirement of resources. The scheduler schedules the task based on the processing capacity, number of queues etc. The scheduler allocates the resources by considering memory, CPU speed, disk capacity etc. The application manager accepts the job from a client and negotiates to execute the first container of the application. The application manager provides the failover mechanism to restart the services, which might have failed due to an application or a hardware failure. Each application manager tracks the status of an individual application. Reference fig. 3.3 shows the YARN architecture.

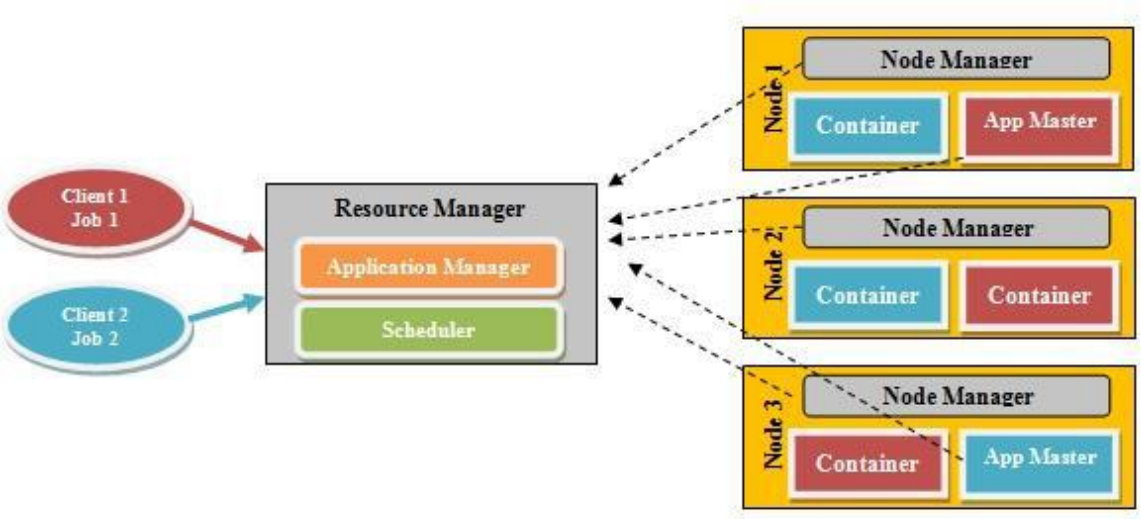

Figure 3.3 YARN Architecture (Hadoop.apache.org, 2019)

3.1.3 MapReduce Programming Model

MapReduce is Google's programming model for parallel processing of distributed data. In distributed processing, it is significant to take data locality into consideration. If the data to be processed is located near, it can reduce the time of transmission and achieve better performance. MapReduce uses this functionality during the map-reduce function. The model uses the distributed data for processing and Hadoop is the most adopted implementation of MapReduce. MapReduce is supported by HDFS and YARN for executing parallel jobs / tasks among multiple datanodes.

MapReduce Model:

MapReduce model consists of two important phases i.e. maps and reduces. As shown in fig.3.4 in "map" phase it takes input as key-value (k, V) pair and produces intermediate key-value pair (k1,V1) → {(k2,V2)} as a result whilst in "reduce" phase it takes a key and a list of the keys and values and generates the final output as key/value (k2; {V2}) → {V3} pair.

In MapReduce each map function will take place on local data and output will be stored in a temporary storage. A master node coordinates the input data only after the input is processed. In the next phase, i.e. the shuffle phase, it randomly generates values assigned and then sorts them according to the assigned values.

Now in reduce phase, the intermediate key value data is processed and the final output is produced.

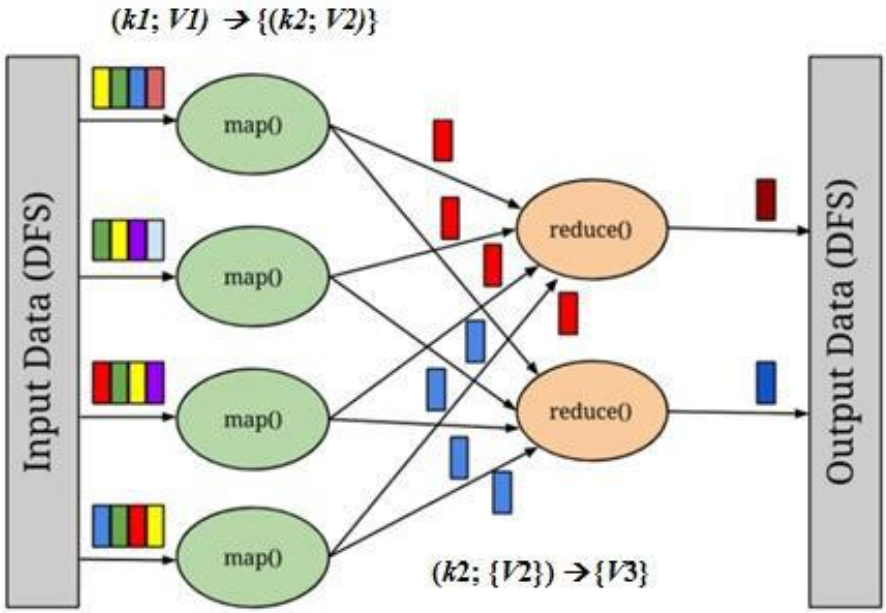

Figure 3.4 MapReduce Model

3.2 HDFS Block Placement Policy

Hadoop uses HDFS block placement policy (Shvachko et al., 2010) for placing data blocks on datanodes. Whenever a client places the request to store data blocks in HDFS, first the dataset is split into blocks according to block size set (default: 64/128 MB) and also the replica of that blocks set is generated in replication factor (default: 3). This policy attempts to place blocks evenly amongst the list of available nodes. In Hadoop, Data locality plays a prominent role in HDFS reliability and performance of MapReduce. Data locality (Team, 2018) relies on moving the computation close to data rather than moving large data blocks to computation. Data locality minimizes network congestion and attains high performance and reliability. "Rack Awareness" is a key concept on which HDFS block placement policy relies. Figure 3.5 demonstrates the default HDFS block placement policy.

1. Hadoop stores data blocks on different datanodes of using placement policy available in HDFS. The policy of placing blocks in Hadoop helps to distribute the data block uniformly amongst the cluster nodes. This policy places the data blocks according to the following approach: Split the files into blocks and

replicates the blocks according to the replication factor that has already been defined in the hdfs-site.xml file.

2. If a request comes from the datanode which is the node of the cluster, the first replica of the block must put on that datanode itself. Otherwise, place it randomly on any datanode of the cluster.

3. The second replica of the block will be placed on nodes of other racks if available or may be placed on the same rack of the first replica.

4. The third replica will be placed on any node of the rack where the second replica is placed.

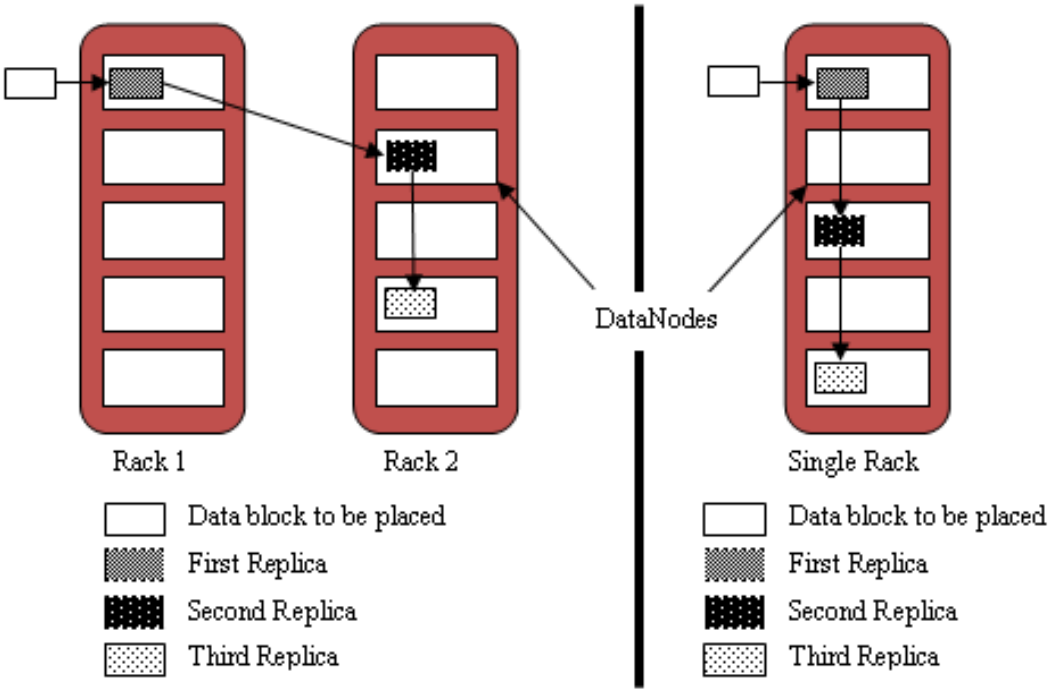

Figure 3.5 HDFS Block Placement Policy

3.2.1 Issues of Default Policy

The performance of Hadoop and MapReduce vastly relies on how well data blocks are distributed and stored on datanodes. HDFS block placement policy uniquely places data blocks but replicas are not evenly distributed. Sometimes, it is important to consider various scenarios like cluster imbalance, network configuration, disk

speed, and many others. A few points that should be considered for improving existing strategy and a few which are already implemented are discussed here.

- **Cluster/Data imbalance:** Default policy places the first copy of the block in the client rack itself and other copies in other racks which sometimes imbalances the cluster. It doesn't consider where an application is going to process. As a result, if racks are geographically distributed it considerably imbalances the cluster.
- **Heterogeneity model:** While placing blocks the current placement policy fails to work efficiently with the heterogeneity of nodes, network configuration, and other resources.
- **Resource awareness:** Default policy doesn't consider processing capability of nodes for placing data blocks. The job may get skewed. It can also consider disk speed, types of storage, memory and other resources for effective data storage and distribution.
- **HDFS balancer:** To deal with cluster imbalance issue, Hadoop has introduced HDFS balancer which balances the cluster based on total distributed storage. However, it also doesn't consider any of the problems discussed above.

3.3 Hadoop Schedulers

YARN uses schedulers to schedule the jobs among multiple shared resources. It supports three scheduling schemes in MapReduce framework: FIFO (First-In, First-Out), Capacity (Hadoop.apache.org, 2018c) and Fair (Hadoop.apache.org, 2018d) scheduler. MapReduce1 (MR1) comes with all three with FIFO as default scheduler, while MR2 comes with capacity and fair scheduler, which can be further configured with delay scheduler to address the locality issue. FIFO scheduler is the simplest and doesn't support priority and size based scheduling. Capacity scheduler guarantees a minimum capacity of resources available to each queue for processing. The latter guarantees fairly share amount of resources among all the queues.

3.3.1 FIFO Scheduler

First-In, First-Out (FIFO) scheduler schedules the oldest job first in a queue. All the available resources will get allocated to the first application/job in a queue and the

cluster won't be shared with other application. Therefore, the job coming next in queue has to wait until the first job finishes its task which leads to poor utilization of cluster. Priority-based and job size based resource allocation is not possible as it doesn't support preemption or application based scheduling. Figure 3.6 shows working of FIFO scheduler.

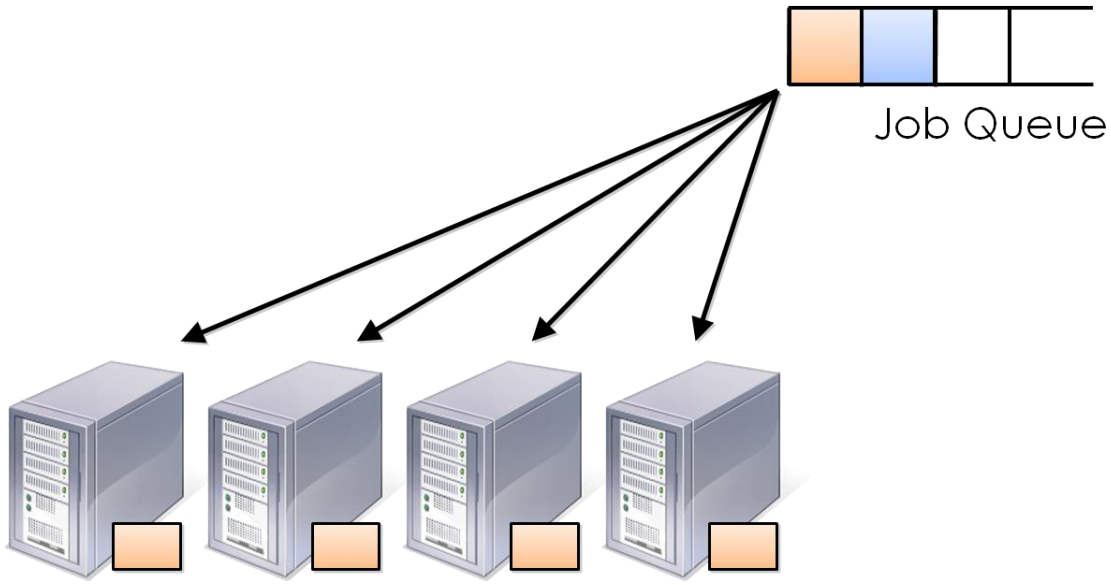

Figure 3.6 FIFO Scheduler (Subhash, G., 2018)

3.3.2 Capacity Scheduler

This is the default scheduler, which comes with MR2 or YARN. The capacity scheduler's configuration supports multiple queues, which can be allocated to multiple users based on tasks or organization. This scheduler is designed with an idea that the same cluster can be rented to multiple organization and resources may be divided among several users. Thus, the organization can divide their resources across multiple departments or users depending upon their tasks or the cluster can also be divided among multiple subsidiary organizations. Each queue can be configured with a fixed portion of resources, which can be soft or hard. Generally, resources are soft having elastic allocation, but they can also be configured for hard approach.

Capacity scheduler makes use of FIFO (First-In First-Out) scheduling if multiple jobs are in the same queue. Suppose a job comes into the queue "A" and if

queue “A” is empty, then it allocates all the resources to the first job. This would utilize more resources than configured capacity of a queue, particularly if queue allocation is elastic and the job requires more resources. When a new job comes in queue “B”, assuming that the first job is still running and using the resources more than its allocated capacity, then tasks of the first job will be killed to free up the resources and allocate those resources to the second job. If another job comes to queue “A” or “B”, the capacity scheduler will process it like FIFO or FIFO with priority. It has many features like capacity guarantee, elasticity, security etc. that can be customized as per requirement. Figure 3.7 shows working of a capacity scheduler.

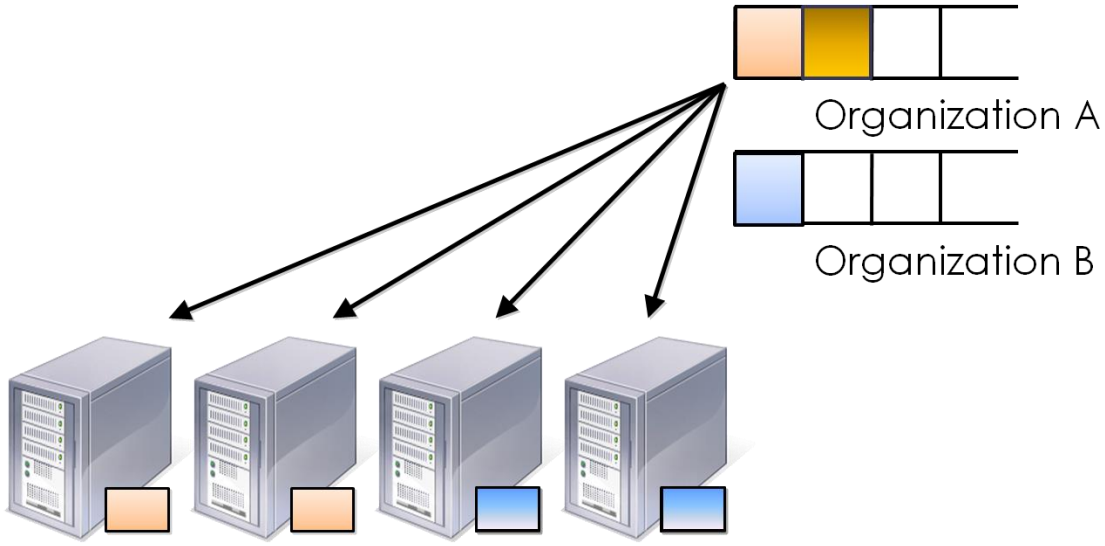

Figure 3.7 Capacity Scheduler (Subhash, G., 2018)

3.3.3 Fair Scheduler

Fair schedulers have a similar queue configuration as discussed in case of capacity scheduler. Jobs would be submitted to the queue, which is termed as a “pool” in case of fair scheduler. Each job will use the allocated resources to their pools. As FIFO approach is followed in capacity scheduler, jobs which come late has to wait till the first job finished or resources are made available. This problem is solved in–fair scheduler. Jobs which have waited in the queue would be picked up and processed simultaneously with the same amount of resources shared by applications that are in the same queue. Fair scheduler supports three scheduling policies that are: FIFO,

Fair, and DRF (Dominant Resource Fairness), to share fair resources within the queues. Figure 3.8 exemplifies working of Fair scheduler.

- **Fair-FIFO:** In FAIR-FIFO scheduling policy, if multiple jobs are in the same queue then resources will be allocated to the job which enters first in the queue, and each job will run serially. However, fair sharing is still being done between the queues.

- **Fair-Fair:** In Fair-Fair scheduling policy, the fair amount of resources will be shared by the jobs that are running in the same queue.

- **Fair-DRF:** Fair-DRF scheduling policy is devised by *Ghodsiet al., 2011*. In FAIR-DRF scheduling policy, DRF evaluates the resources shared by each user, finds out the maximum resource utilized and calls it a dominant resource of the user. The idea is to make uniform resource sharing among the users through equalizing the resources like CPU and Memory.

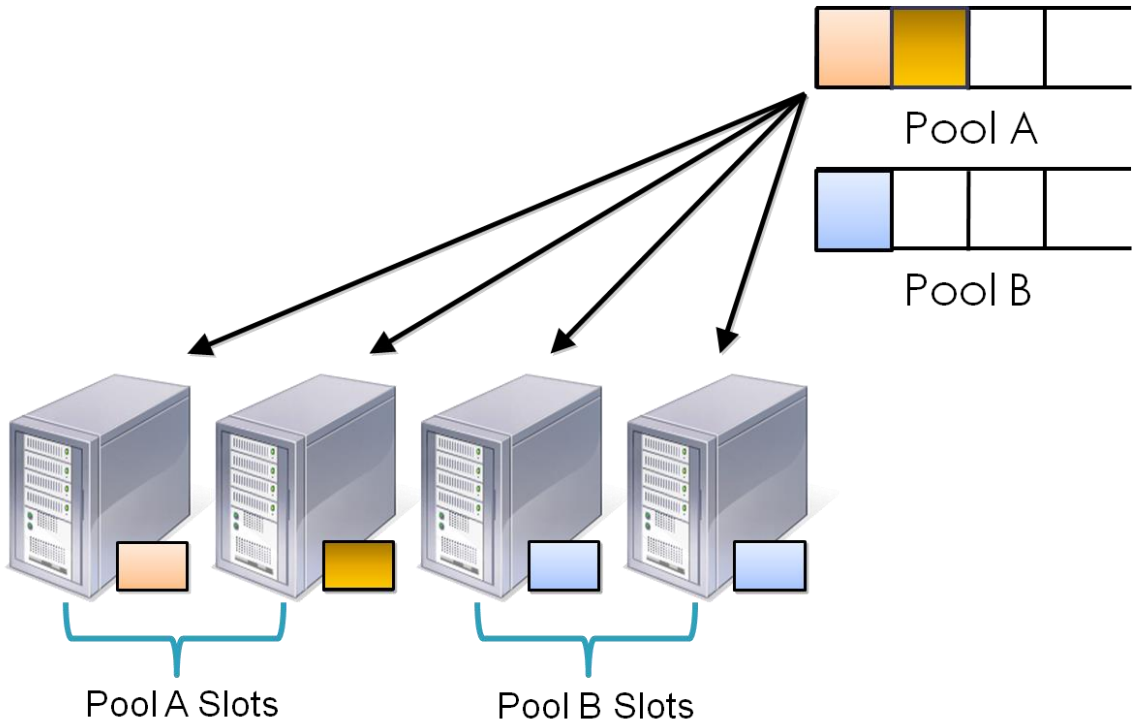

Figure 3.8 Fair Scheduler (Subhash, G., 2018)

3.4 YARN Node Label

YARN Node label (Hadoop.apache.org, 2018e) allows partitioning the single cluster into multiple subclusters. Using this concept node can be marked with meaningful labels i.e. Nodes with higher processing capability may be labelled as “high_cpu” and with high memory may be labelled as “high_mem”. Using node labels job containers can be specified to run on specific nodes only. For example, in Hadoop WordCount a job which requires a lot of computation is considered a CPU intensive job where as TeraSort which requires a lot of memory is considered as an I/O intensive job. Therefore, “high_cpu” label nodes can be provided to WordCount job and “high_mem” label nodes to TeraSort job so that their containers will run on those labelled nodes only.

3.4.1 YARN Node Label Working

A product label helps the consumer to get distinct information about the product. Based on label information a consumer decides whether to buy a product or not. In the same way, using node label information YARN decides where it should run a job. For each job, multiple task containers are allocated by YARN that run only on nodes with the specified node label. Two types of node labels can be assigned viz “Exclusive” and “Non-exclusive”. “Exclusive” node label allows applications to run containers only if it is requested by them, while in “Non-exclusive” node label resources can be shared and it allows applications to run containers even if no label is specified.

Resources are managed by YARN using the hierarchy of queues. Queues can be configured using various scheduling policies discussed above. Here fig. 3.9 shows how the nodes are configured to use queues for allocating containers for the job. There are 3 queues configured named as “A”, “B” and “C” and all nodes are partitioned into 3 parts.

1. Partition X (Exclusive Node Label-X) –As shown in fig 3.9 all nodes of partition X are labelled as node label “X” Exclusive. Only queue “A” can access the partition X as it is an Exclusive node label with capacity configured as 100%.

2. Partition Y (Non-exclusive Node Label-Y) – All nodes of Partition Y are labelled as node label "Y" Non-exclusive. Queue "A" and "B" share the partition with a capacity of 50% each.

3. Default Partition – No labels are assigned to the nodes that come into this partition. Queue "A", "B", and "C" have access to these nodes in a capacity of 40%, 30%, and 30% respectively.

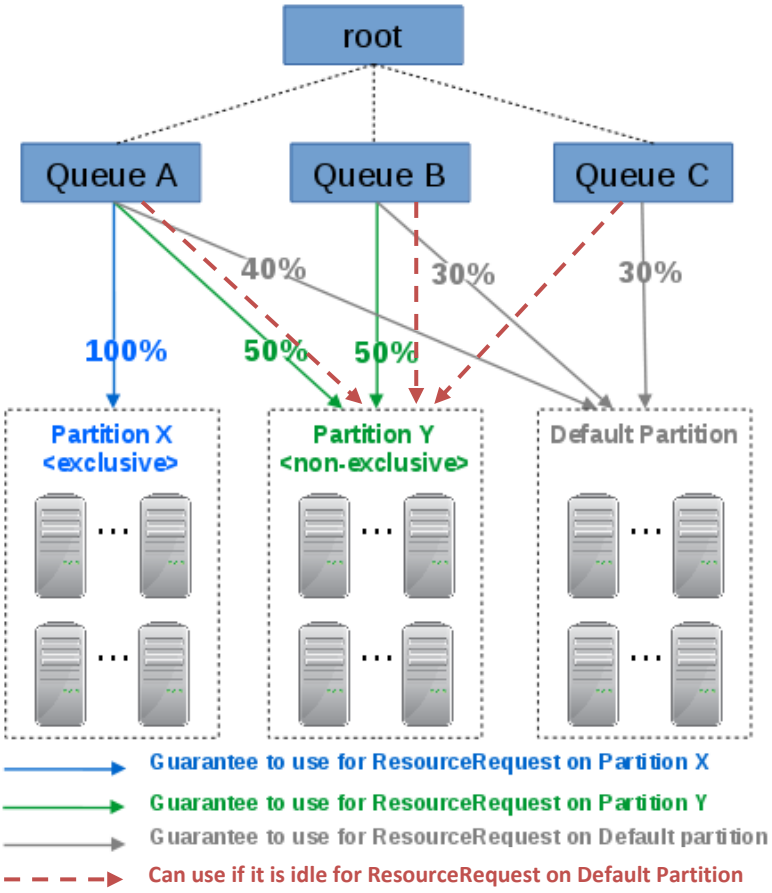

Figure 3.9 Node Label Configuration for Queues (Zhang and Zhang, 2018)

Using above configurations four applications are submitted in queue "A" and "C" to see the effect of node label to run containers. As shown in fig 3.10 "user-1" has submitted 3 applications and "user-2" has submitted 1 application for processing.Application-1 and application-2 are submitted by user-1 in queue "A" using node label "X" and "Y" respectively. Containers of application-1 and application-2 are allocated on partition-X and partition-Y respectively.Application-3 and application-4 are submitted by user-1 and user-2 respectively without

specifying anode label.Application-3 is submitted in queue “A” and application-4 is submitted in queue “C”. Partition-Y is “Non-exclusive”; hence if nodes of that partition are idle, they can be utilized by applications requesting default partition. Therefore, containers of application-3 and application-4 are allocated to partition-Y and default partition as shown in fig 3.10.

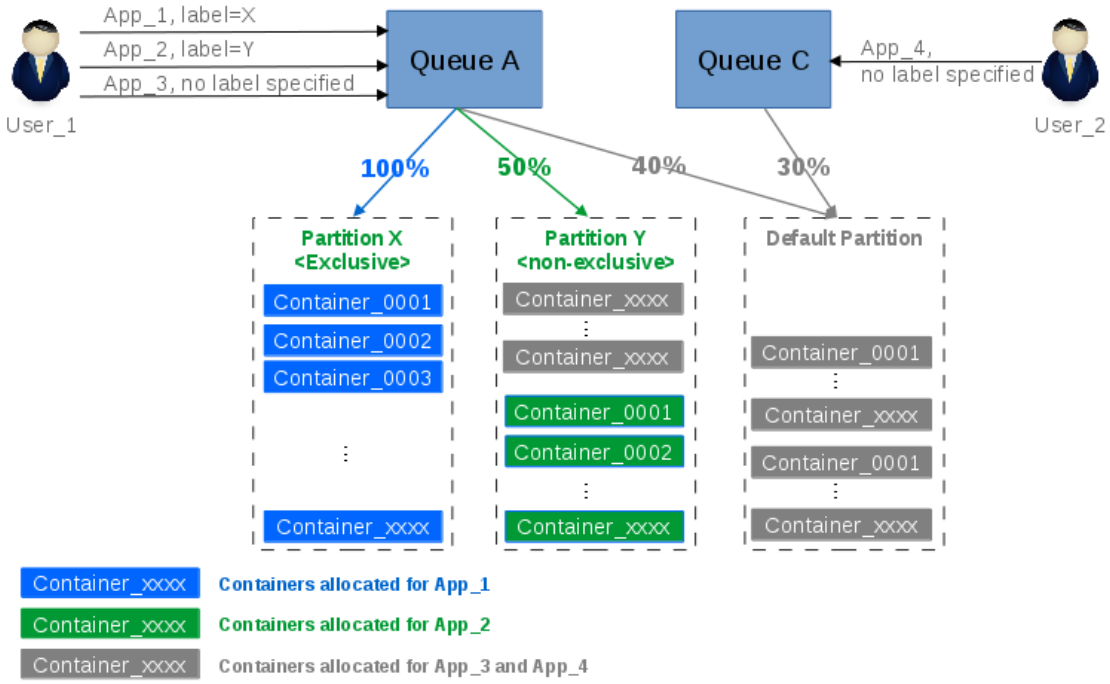

Figure 3.10 Application Submission to Queues (Zhang and Zhang, 2018)

3.4.2 Issues of Node Label

As discussed in the above example node label allows us to select where to place application containers. Node label selection for putting job is equally critical because if nodes selected by using node label for job processing do not leverage cluster effectively. It degrades the application and job performance. Sometimes, if a partition is selected as “Exclusive” and data blocks are not present at that location to run the container then it requires moving data block all the way from other nodes or racks. Moving data block where containers are running is a very exhausting process as it degrades the performance of Hadoop drastically.

3.5 Challenges in Hadoop

Hadoop cluster configuration is challenging since Hadoop framework is a complex distributed environment that involves a combination of hardware and software which affects the performance of Big Data application. On the hardware side, HDFS performance relies on data storage, access and data transfer between datanodes. Datanode hardware specifications like processors, memory, and storage space also play a major role in it. On the software side, Hadoop can be customized and tuned to achieve better performance. Various challenges that must be taken into account to achieve better Hadoop performance are discussed below:

- **Heterogeneous Environment:** Hadoop is specifically designed for homogeneous nodes only. In today's era of computing, we cannot imagine having a cluster made up of homogeneous nodes only. Data locality variation on a heterogeneous cluster is the biggest challenge to answer. To achieve better Hadoop performance on a heterogeneous cluster is one of the major challenges to exhibit.

- **HDFS Block Placement Policy:** Hadoop default HDFS block placement strategy can be improvised, since it does not distribute blocks uniformly based on the processing capability of nodes. Default policy requires major changes specifically in heterogeneous Big Data application processing.

- **Load Balancing:** Though Hadoop handles load balancing automatically, it is also one of the major concern**s** for the researchers and scientists. Sometimes, the load balancing issue occurs as data is not distributed evenly, so finished tasks have to wait for running tasks. There is a need of technique which can avoid or mitigate the effect of data skew.

- **Resource Awareness:** HDFS and MapReduce do not consider the resource capability while storing and putting job request to datanodes. It can achieve better performance if the system is resource-aware (e.g. processor, number of cores, memory).