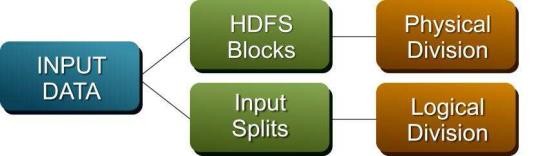


**Map Stage:** The map or mapper’s job is to process the input data. Generally the input data is in the form of file or directory and is stored in the Hadoop file system (HDFS). The input file is passed to the mapper function line by line. The mapper processes the data and creates several small chunks of data that is in the form of key value pairs.

**Reduce Stage:** This stage is the combination of the Shuffle stage and the Reduce stage. The Reducer’s job is to process the data that comes from the mapper such as consolidation, aggregation or sorting. After processing, it produces a new set of output, which will be stored in the HDFS.

# Input splits



The way HDFS has been set up, it breaks down very large files into large blocks (for example, measuring 128MB), and stores three copies of these blocks on different nodes in the cluster.

MapReduce data processing is driven by this concept of input splits. The number of input splits that are calculated for a specific application determines the number of mapper tasks. Each of these mapper tasks is assigned, where possible, to a slave node where the input split is stored. One line represents one record. Now, remember that the block size for the Hadoop cluster is 64MB, which means that the light data files are broken into chunks of exactly 64MB.

Do you see the problem? If each map task processes all records in a specific data block, what happens to those records that span block boundaries? File blocks are exactly 64MB (or whatever you set the block size to be), and because HDFS has no conception of what’s inside the file blocks, it can’t gauge when a record might spill over into another block.

To solve this problem, Hadoop uses a logical representation of the data stored in file blocks, known as input splits. When a MapReduce job client calculates the input splits, it figures out where the first whole record in a block begins and where the last record in the block ends.

In cases where the last record in a block is incomplete, the input split includes location information for the next block and the byte offset of the data needed to complete the record. The above figure shows this relationship between data blocks and input splits.

# Hadoop Version1 Storage & Processing Daemons

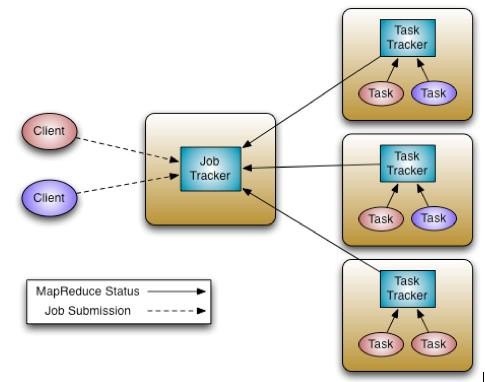
Below flow is self-explanatory.

# Hadoop Version 1 Processing Daemons:

**Job Tracker (Per Cluster) :**

***JobTracker*** is master processing software daemon responsible **Resource allocation, resource administration, scheduling and monitoring** of jobs by taking in requests from a client and assigning *TaskTrackers* with tasks to be performed.

The *JobTracker* tries to assign tasks to the *TaskTracker* on the *DataNode* where the data is locally present (Data Locality).



# Roles of Job Tracker:

Job Tracker is the daemon service for submitting and tracking MapReduce jobs in Hadoop. There is only One Job Tracker process run on any hadoop cluster. Job Tracker runs on its own JVM process. In a typical production cluster its run on a separate machine. Each slave node is configured with job tracker node

location. The JobTracker is single point of failure for the Hadoop MapReduce service. If it goes down, all running jobs are halted. JobTracker in Hadoop performs following actions…

The JobTracker is the service within Hadoop that farms out [MapReduce](http://wiki.apache.org/hadoop/MapReduce) tasks to specific nodes in the cluster, ideally the nodes that have the data, or at least are in the same rack.

1. Client applications submit jobs to the Job tracker.
2. The JobTracker talks to the [NameNode](http://wiki.apache.org/hadoop/NameNode) to determine the location of the data.
3. The JobTracker locates [TaskTracker](http://wiki.apache.org/hadoop/TaskTracker) nodes with available slots at or near the data.
4. The JobTracker submits the work to the chosen [TaskTracker](http://wiki.apache.org/hadoop/TaskTracker) nodes.
5. The [TaskTracker](http://wiki.apache.org/hadoop/TaskTracker) nodes are monitored. If they do not submit heartbeat signals often enough, they are deemed to have failed and the work is scheduled on a different [TaskTracker](http://wiki.apache.org/hadoop/TaskTracker).
6. A [TaskTracker](http://wiki.apache.org/hadoop/TaskTracker) will notify the JobTracker when a task fails. The JobTracker decides what to do then: it may resubmit the job elsewhere, it may mark that specific record as something to avoid, and it may may even blacklist the [TaskTracker](http://wiki.apache.org/hadoop/TaskTracker) as unreliable.
7. When the work is completed, the JobTracker updates its status.
8. Client applications can poll the JobTracker for information.

**TaskTracker (Per Node)**

1. A **TaskTracker** is a slave node daemon in the cluster that accepts tasks (Map, Reduce and Shuffle operations) from a JobTracker.
2. There is only One Task Tracker process run on any hadoop data node.
3. Task Tracker runs on its own JVM process.
4. Every TaskTracker is configured with a set of slots; these indicate the number of tasks that it can accept. The TaskTracker starts a separate JVM processes to do the actual work (called as Task Instance) this is to ensure that process failure does not take down the task tracker.
5. The TaskTracker monitors these task instances, capturing the output and exit codes. When the Task instances finish, successfully or not, the task tracker notifies the JobTracker.
6. The TaskTrackers also send out heartbeat messages to the JobTracker, usually every 5 seconds, to reassure the JobTracker that it is still alive. These message also inform the JobTracker of the number of available slots, so the JobTracker can stay up to date with where in the cluster work can be delegated.

**Limitation in Version1 with respect to processing**

# Job tracker is overburden

JobTracker runs on single machine doing several task like

* Resource allocation and administration.
* Job and task scheduling.
* Monitoring.

Although there are so many machines (Data Node) available; they are not getting used. This limits scalability of JobTracker.

Responsibilities of the JobTracker (Hadoop 1.0) mainly included the following:

* + Managing the computational resources in terms of map and reduce slots
  + Scheduling submitted jobs
  + Monitoring the executions of the TaskTrackers
  + Restarting failed tasks
  + Performing a speculative execution of tasks
  + Calculating the Job Counters

Clearly, the JobTracker alone does a lot of tasks together and is overloaded with lots of work.

# Job tracker is single point of failure

In Hadoop 1.0, JobTracker is single Point of availability. This means if JobTracker fails, all jobs must restart as there is no mechanism to maintain/store the interim state of the JT progress to start it from where the job is left.

# Fixed slots for Map & Reduce tasks

In Hadoop 1.0, there is concept of predefined number of map slots and reduce slots for each TaskTrackers. Resource Utilization issues occur because maps slots might be ‘full’ while reduce slots is empty (and vice-versa). Here the compute resources (DataNode) could sit idle which are reserved for

Reduce slots even when there is immediate need for those resources to be used as Mapper slots. Due to

this fixed slot limitation version 1 can scale up to 4,000 nodes or 40000 tasks.

In map-reduce v1 mapreduce.tasktracker.map.tasks.maximum and mapreduce.tasktracker.reduce.tasks.maximum are used to configure number of map slots and reduce slots accordingly in mapred-site.xml.

# MR is only Framework supported:

In Hadoop 1.0, Job tracker was tightly integrated with MapReduce and only supporting application that obeys MapReduce programming framework can run on Hadoop.

Let’s try to understand point 4 in more detail.

MapReduce works on batch-driven data analysis, where the input data is partitioned into smaller batches that can be processed in parallel across many machines in the Hadoop cluster. But MapReduce, while powerful enough to express many data analysis algorithms, is not always the optimal choice of

programming paradigm. It‘s often desirable to run other computation paradigms in the Hadoop cluster – here are some examples.

* **Problem in performing real-time analysis:** MapReduce is batch driven. What if I want to do perform real time analysis instead of batch-processing (where results is available after several hours).

There are many applications which need results in real time like fraud detection algorithm. There are real time engines like Apache Storm which can work better in this case. But in Hadoop 1.0, due to tight coupling these engines cannot run independently.

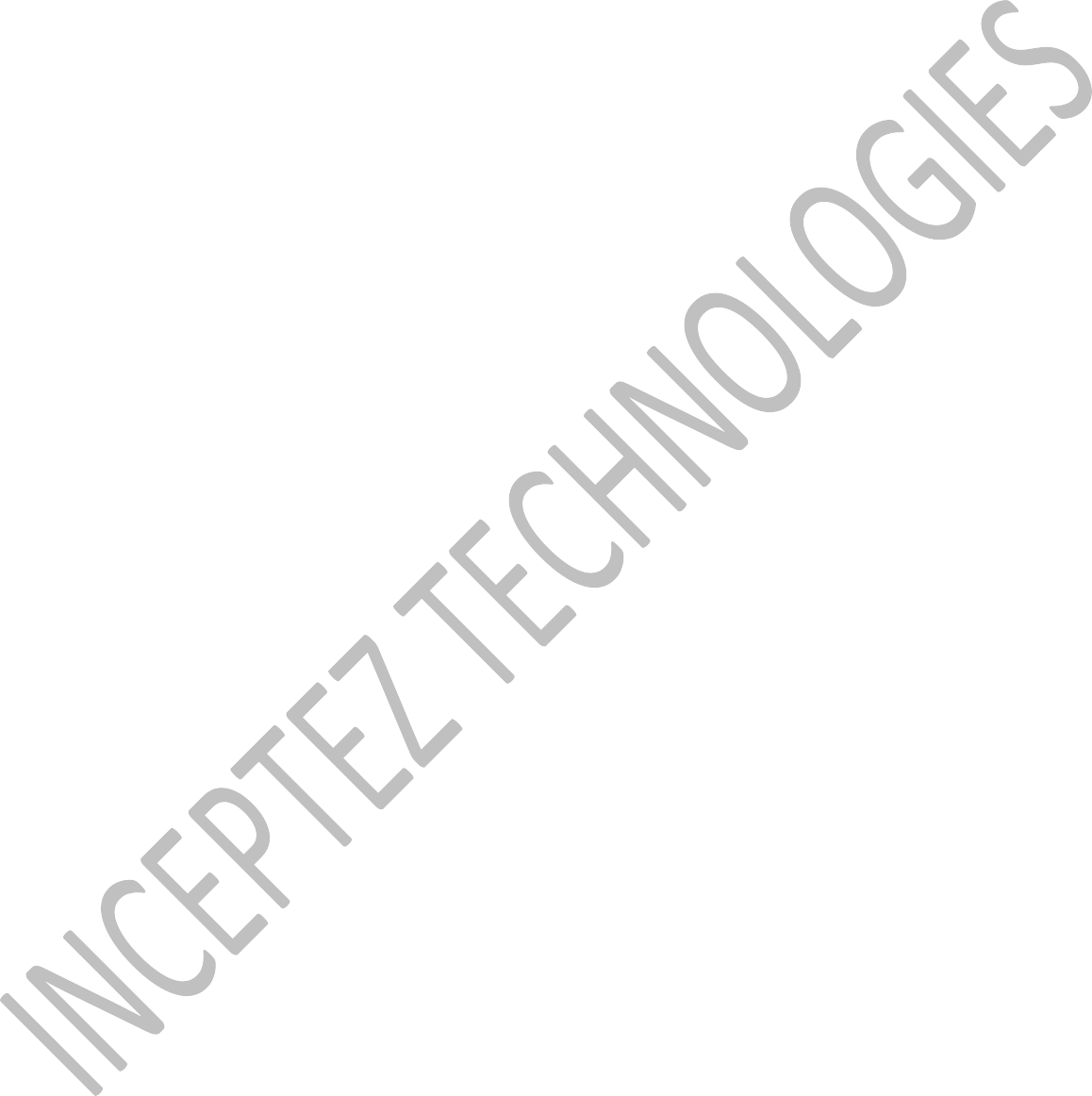
* **Problem in running Message-Passing approach:** It is a stateful process that runs on each node of a distributed network. The processes communicate with each other by sending messages, and alter their state based on the messages they receive. This is not possible in MapReduce.
* **Problem in running Ad-hoc query:** Many users like to query their big data using SQL. Apache Hive can execute a SQL query as a series of MapReduce jobs, but it has shortcomings in terms of performance. Recently, some new approaches such as Apache Tajo , Facebook's Presto and Cloudera's Impala drastically improve the performance, but they require to run services in other form than MapReduce form.

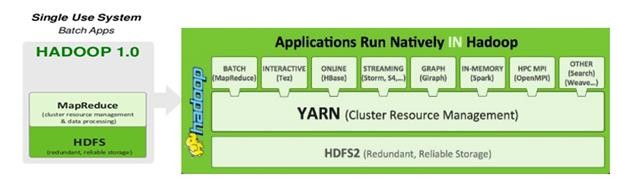
It is not possible to run all such non Map Reduce jobs on Hadoop Cluster. Such jobs have to "disguise" themselves as mappers and reducers in order to be able to run on Hadoop 1.0.

# Backward compatibility issue

Older version of Map reduces MR1 could not run in the latest MR1 versions, whereas YARN is backward compatible to run older versions.

# Transition of Hadoop Version 1 to 2

**YARN** – Yet Another Resource Negotiator is a resource manager that was created by separating the processing engine and resource management capabilities of MapReduce as it was implemented in Hadoop 1. YARN is often called the Data operating system of Hadoop because it is responsible for managing and monitoring workloads, maintaining a multi-tenant environment, implementing security controls, and managing high availability features of Hadoop.



Like an operating system on a server, YARN is designed to allow multiple, diverse user applications to run on a multi-tenant platform. In Hadoop 1, users had the option of writing MapReduce programs in Java, in Python, Ruby or other scripting languages using streaming, or using Pig, a data transformation language. Regardless of which method was used, all fundamentally relied on the MapReduce processing model to run.

YARN supports multiple processing models in addition to MapReduce. One of the most significant benefits of this is that we are no longer limited to working the often I/O intensive, high latency MapReduce framework. This advance means Hadoop users should be familiar with the pros and cons of the new processing models and understand when to apply them to particular use cases.

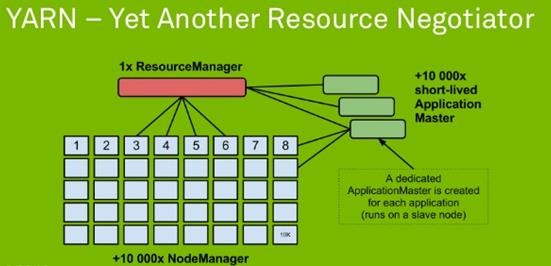
**Architectural Changes**

Following two figures represents the architectural changes of the version 1 daemons (Job tracker and Task tracker) are replaced by Resource manager, Application Master, Node manager and Containers.

In the proposed new YARN architecture, the J**ob Tracker** is split into two different daemons called ***Resource Manager*** and ***Node Manager and Task Tracker*** *is split into Appmaster and*

*container****.*** The resource manager only manages the allocation of resources to the different applications submitted and maintains the *scheduler which just takes care of the scheduling jobs without worrying about any monitoring or status updates*. Different resources such as *memory, cpu time, network*

*bandwidth etc. are put into one unit called the* ***Container****.* ***Node Manager*** *will manage all containers by creating it for Resource manager to launch AppMasters that executes the actual tasks by launching more containers*. There are different ***AppMasters*** running on different nodes which talk to a number of these resource containers and accordingly update the Node Manager with the monitoring/status details.



**YARN Components**

# ResourceManager (RM)

1. **ApplicationsManager**
2. **NodeManager (NM)**
3. **ApplicationMaster (AM)**
4. **Container**
   1. **ResourceManager (RM)**

The ResourceManager (RM) is the key service offered in YARN. Clients can interact with the framework using ResourceManager. ResourceManager is the master for all other daemons available in the framework.

**ResourceManager** has two major components, **Scheduler and ApplicationsManager**

# Scheduler

1. The Scheduler is responsible for allocating resources to the various running applications subject to familiar constraints of capacities, queues etc.
2. It is a pure scheduler since it does not monitor or track the status of the application instead it purely performs its scheduling function based the resource requirements of the applications
3. It schedules the resources depending on the resource “Container”
4. The Scheduler has the pluggable policy plug-in which is responsible for partitioning the cluster resources among the various queues, applications etc, for example a) CapacityScheduler, b)

FairScheduler

# ApplicationsManager

1. Applications Manager is responsible for accepting job-submissions.
2. Assigning the first container for executing the application specific Application-Master.
3. Provides the service for restarting the Application-Master container on failure.

# NodeManager (NM)

1. Node-Manager is responsible for Container management.
2. Monitoring container resource usage (like cpu, memory, disk, network).
3. Reporting to the Resource-Manager/Scheduler.

# ApplicationMaster (AM)

1. Application Master is responsible for negotiating resources with the ResourceManager and for working with the NodeManagers to start the containers.
2. Application-Master is responsible for negotiating appropriate resource containers from the Scheduler.
3. Tracking the status and monitoring progress for applications running in each containers under the Application-Master.

# Container

Resource Container incorporates elements such as memory, cpu, disk, network, Command line to launch the process within the container, Environment variables and Local resources necessary on the machine prior to launch, such as jars, shared-objects, auxiliary data files, Security-related tokens etc.

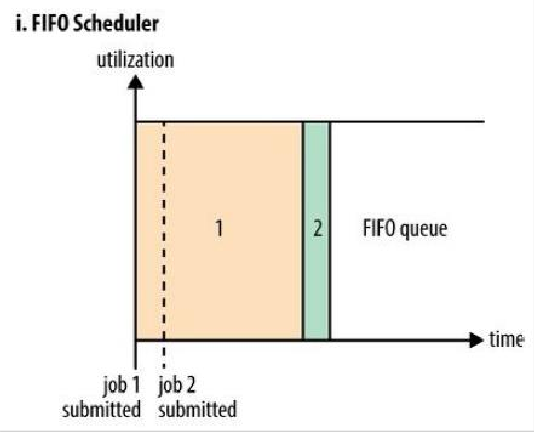
# Schedulers:

There are three schedulers are available in YARN: the FIFO, Capacity, and Fair Schedulers. The

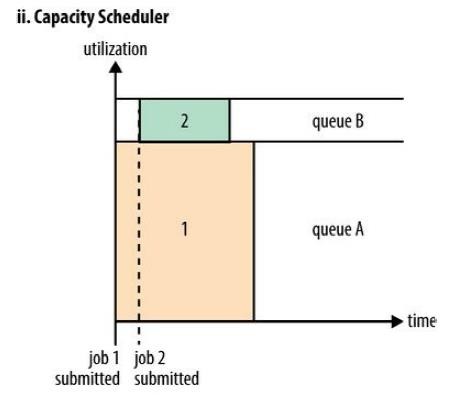
**FIFO Scheduler** places applications in a queue and runs them in the order of submission (first in, first out). Requests for the first application in the queue are allocated first; once its requests have been satisfied, the next application in the queue is served, and so on.

The FIFO Scheduler has the merit of being simple to understand and not needing any

configuration, but it’s not suitable for shared clusters as we none of the critical applications could not be prioritized that results in missing SLA. Large applications will use all the resources in a cluster, so each application has to wait its turn. On a shared cluster it is better to use the Capacity Scheduler or the Fair Scheduler. Both of these allow long runningjobs to complete in a timely manner, while still allowing users who are running concurrent smaller ad hoc queries to get results back in a reasonable time.

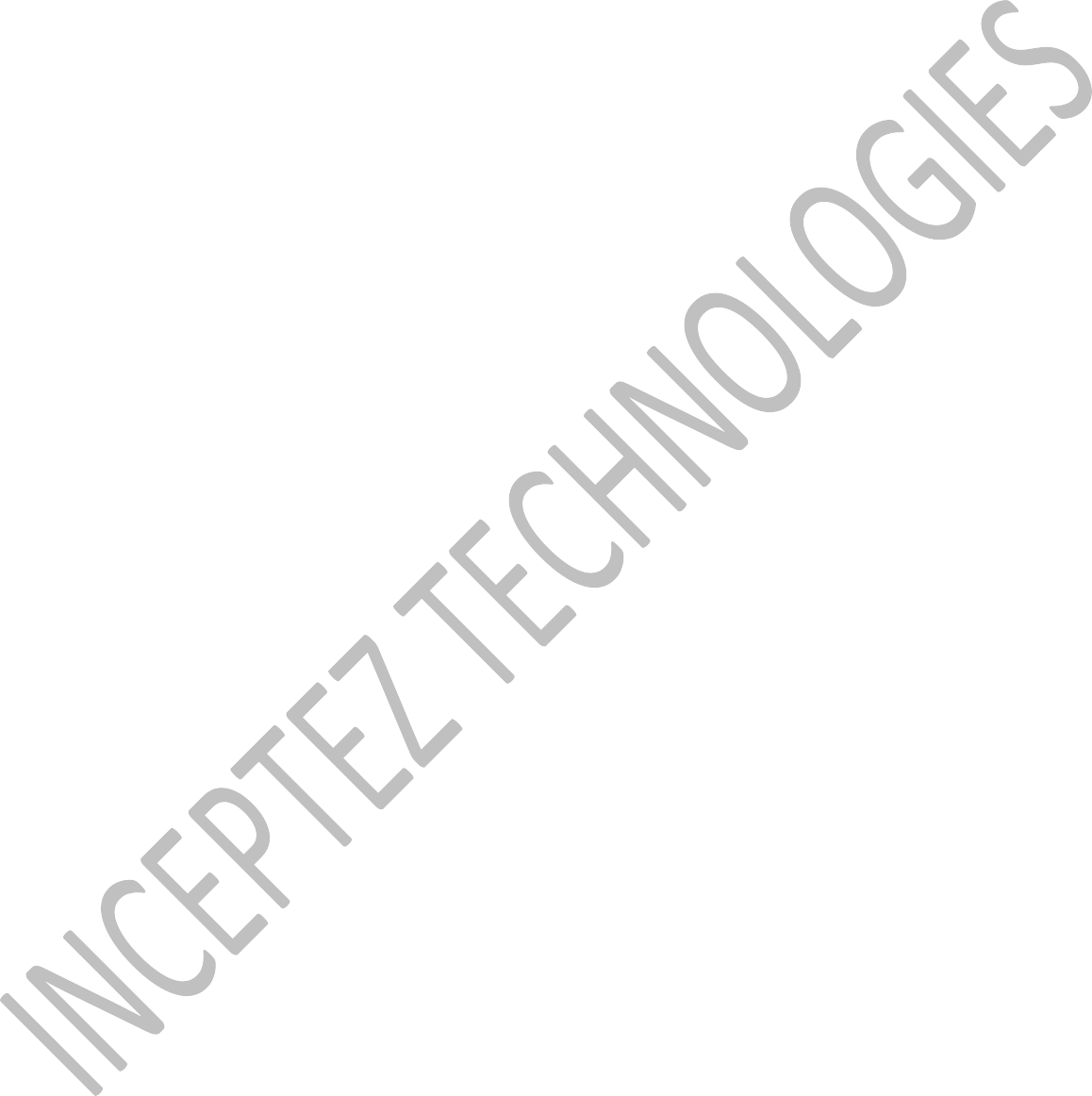


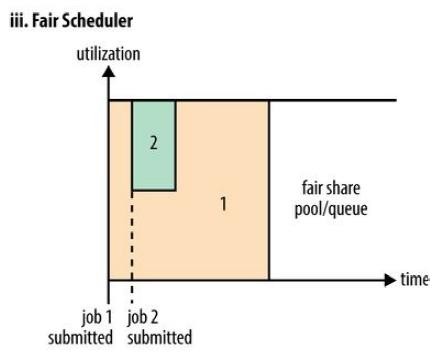
**CapacityScheduler** is designed to allow sharing a large cluster while giving each organization capacity guarantees. The central idea is that the available resources in the Hadoop cluster are shared among multiple organizations who collectively fund the cluster based on their computing needs. There is an added benefit that an organization can access any excess capacity not being used by others. This provides elasticity for the organizations in a cost-effective manner. The primary abstraction provided by the CapacityScheduler is the concept of *queues*. These queues are typically setup by administrators to reflect the economics of the shared cluster. To provide further control and predictability on sharing of resources, the CapacityScheduler supports *hierarchical queues* to ensure resources are shared among the sub-queues of an organization before other queues are allowed to use free resources, there-by providing *affinity* for sharing free resources among applications of a given organization.



Sharing clusters across organizations necessitates strong support for multi-tenancy since each organization must be guaranteed capacity and safe-guards to ensure the shared cluster is impervious to single rouge application or user or sets thereof. The CapacityScheduler provides a stringent set of limits to ensure that a single application or user or queue cannot consume disproportionate amount of resources in the cluster. Also, the CapacityScheduler provides limits on initialized/pending applications from a single user and queue to ensure fairness and stability of the cluster.

**Fair Scheduler** is a reasonable way to share a cluster between a number of users. Finally, fair sharing can also work with app priorities - the priorities are used as weights to determine the fraction of total resources that each app should get.

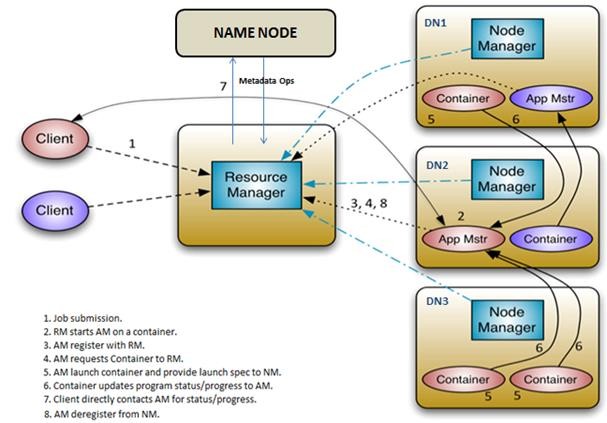
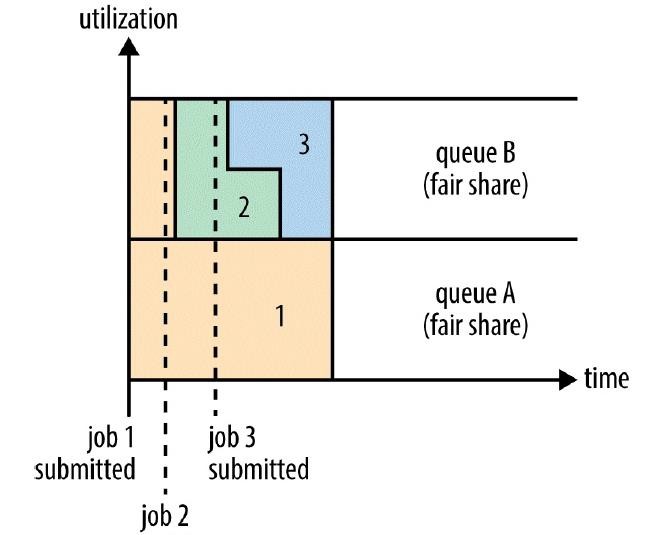
The scheduler organizes apps further into “queues”, and shares resources fairly between these queues. By default, all users share a single queue, named “default”. If an app specifically lists a queue in a container resource request, the request is submitted to that queue. It is also possible to assign queues based on the user name included with the request through configuration. Within each queue, a scheduling policy is used to share resources between the running apps. The default is memory-based fair sharing, but FIFO and multi-resource with Dominant Resource Fairness can also be configured.



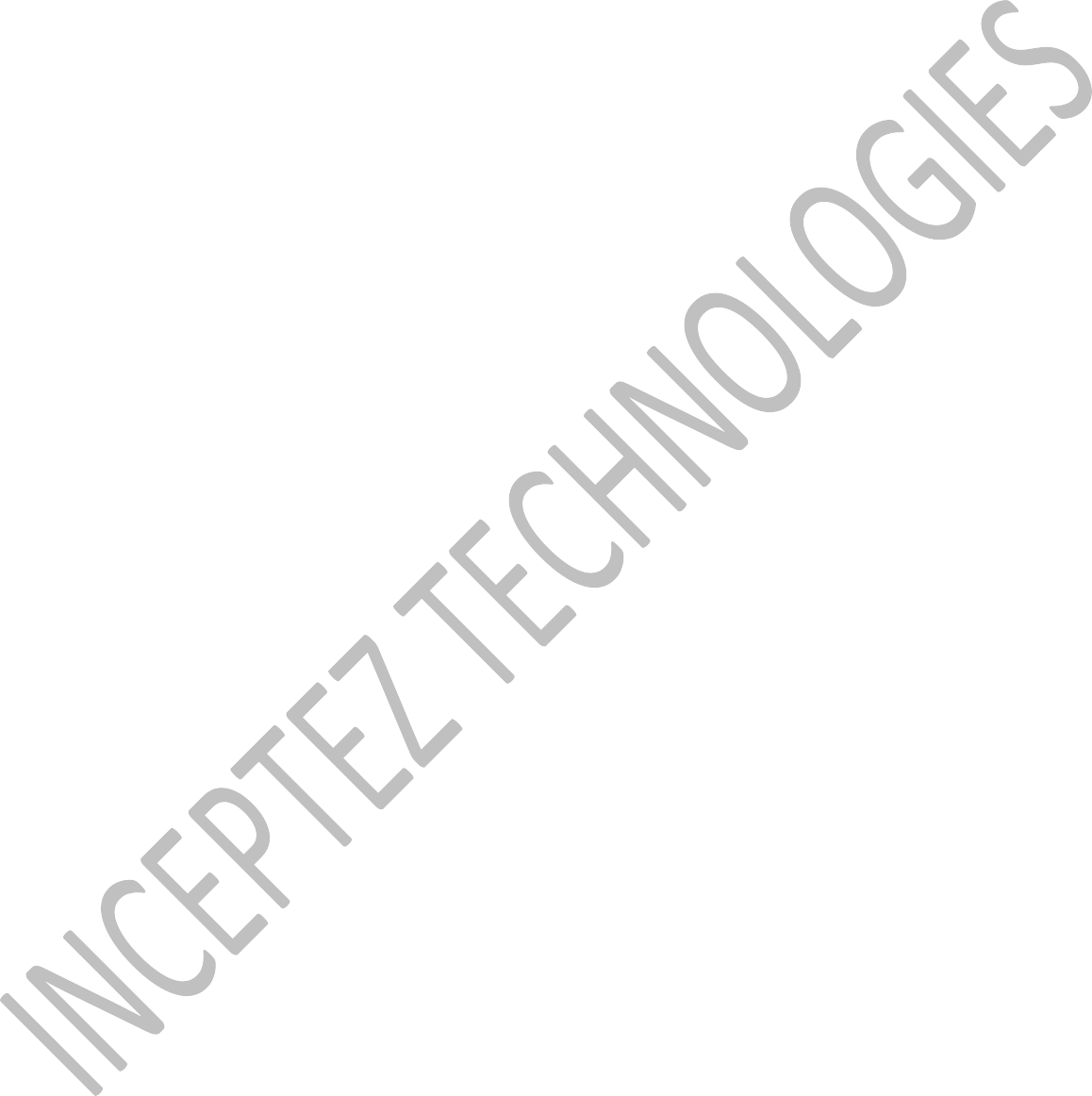
Queues can be arranged in a hierarchy to divide resources and configured with weights to share the cluster in specific proportions.

In addition to providing fair sharing, the Fair Scheduler allows assigning guaranteed minimum shares to queues, which is useful for ensuring that certain users, groups or production applications always get sufficient resources. When a queue contains apps, it gets at least its minimum share, but when the queue does not need its full guaranteed share, the excess is split between other running apps. This lets the scheduler guarantee capacity for queues while utilizing resources efficiently when these queues don’t contain applications.The Fair Scheduler lets all apps run by default, but it is also possible to limit the number of running apps per user and per queue through the config file. This can be useful when a user must submit hundreds of apps at once or in general to improve performance if running too many apps at once would cause too much intermediate data to be created or too much context-switching.

Limiting the apps does not cause any subsequently submitted apps to fail, only to wait in the scheduler’s queue until some of the user’s earlier apps finish.



**YARN Architecture**

*Let’s walk through an application execution sequence (steps are illustrated in the diagram):*

1. A client program *submits* the application, including the necessary specifications to *launch the application-specific ApplicationMaster* itself.
2. The ResourceManager assumes the responsibility to negotiate a specified container in which to start the ApplicationMaster and then *launches* the ApplicationMaster.
3. The ApplicationMaster, on boot-up, *registers* with the ResourceManager – the registration allows the client program to query the ResourceManager for details, which allow it to directly communicate with its own ApplicationMaster.
4. During normal operation the ApplicationMaster negotiates appropriate resource containers to RM.
5. On successful container allocations, the ApplicationMaster launches the container by providing the container launch specification to the NodeManager. The launch specification, typically, includes the necessary information to allow the container to communicate with the ApplicationMaster itself.
6. The application code executing within the container then provides necessary information (progress, status etc.) to its ApplicationMaster via an *application-specific protocol*.
7. During the application execution, the client that submitted the program communicates directly with the ApplicationMaster to get status, progress updates etc. via an application-specific protocol.
8. Once the application is complete, and all necessary work has been finished, the ApplicationMaster deregisters with the ResourceManager and shuts down, allowing its own container to be repurposed.

**UBER mode**

In normally mappers and reducers will run by ResourceManager (RM), RM will create separate container for mapper and reducer. uber configuration, will allow to run mapper and reducers in the same process as the ApplicationMaster (AM) runs in the same container. **Uber jobs** are jobs that are executed within the MapReduce ApplicationMaster. Rather then communicate with RM to create the mapper and reducer containers. The AM runs the map and reduce tasks within its own process and avoided the overhead of launching and communicate with remote containers. If you have a small dataset, want to run MapReduce on small amount of data. Uber configuration will help you out, by reducing additional time that MapReduce normally spends mapper and reducers phase.

# JOB SUBMISSION IN YARN

* 1. Clients submit Map Reduce job by interacting with Job objects, clients runs in its own JVM.
  2. Job’s code interacts with Resource Manager to acquire application metadata such as application id.
  3. Job’s code move all job related resources to HDFS to make them available for rest of the job.
  4. Job’s code submits the application to Resource Manager.
  5. Resource Manager chooses a Node manager with available resources and requests a container for MRAppMaster.
  6. Node manager allocates container for MRAppMaster; MRAppMaster will execute and co- ordinate Map Reduce Job.
  7. MRAppMaster grabs required resources from HDFS, such as input splits; these resources were copied in step 3.
  8. MRAppMaster negotiates with Resource Manager for available resources; Resource Manager will select Node Manager that has the most resources and the data blocks.
  9. MRAppMaster tells selected Node Manager to start Map and Reduce tasks.
  10. Node Manager creates YARN Child containers that will coordinate and run tasks.
  11. YARN Child acquires job resources from HDFS that will be required to execute Map and Reduce tasks.
  12. YARN Child executes Map and Reduce Tasks.