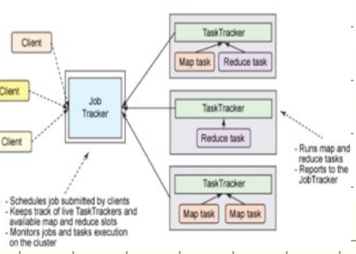
YARN : yet another resource negotiator

- Recall
  - Job – the entire map reduce application
  - Task – Individual mappers/reducers
- How do we
  - Allocate resources – determine which nodes will run the jobs
  - Monitor the tasks – start new tasks or restart failed/slow tasks
  - Monitor the overall state of the job?

In hadoop 1

- Job Tracker
  - Manage Cluster resources
  - Job scheduling
- Task Tracker
  - One per task
  - Manage the task
- Fault Tolerance, Cluster resource management and scheduling handled by JobTracker



Client
Client
Client
Job Tracker
TaskTracker
Map task
Reduce task
TaskTracker
Reduce task
TaskTracker
Map task
Map task
- Schedules job submitted by clients
- Keeps track of live TaskTrackers and available map and reduce slots
- Monitors jobs and tasks execution on the cluster
- Runs map and reduce tasks
- Reports to the JobTracker

Limits scalability
- Job tracker runs on a single machine and is responsible for cluster management, scheduling and monitoring

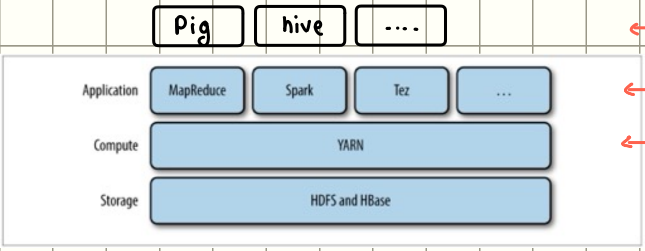
Availability
- JobTracker is the single point of availability/failure

Resource utilization problems
- Predefined #map/reduce slots. Utilization issues because map slots may be full but reduce slots are free.

Limitation in running MR applications
- Tightly integrated with Hadoop. Only MR apps can run. Can't coexist with other applications.

YARN

- Hadoop's cluster resource management system
- Provides API to applications, users don't usually directly work with yarn.



more applications built on MR, spark etc...

users use

yarn manages resources

Anatomy of YARN application run

YARN runs 2 daemons

1] Resource Manager (one per cluster) : manages resources across the cluster

2] Node Managers (runs on all nodes) : launches & monitors containers

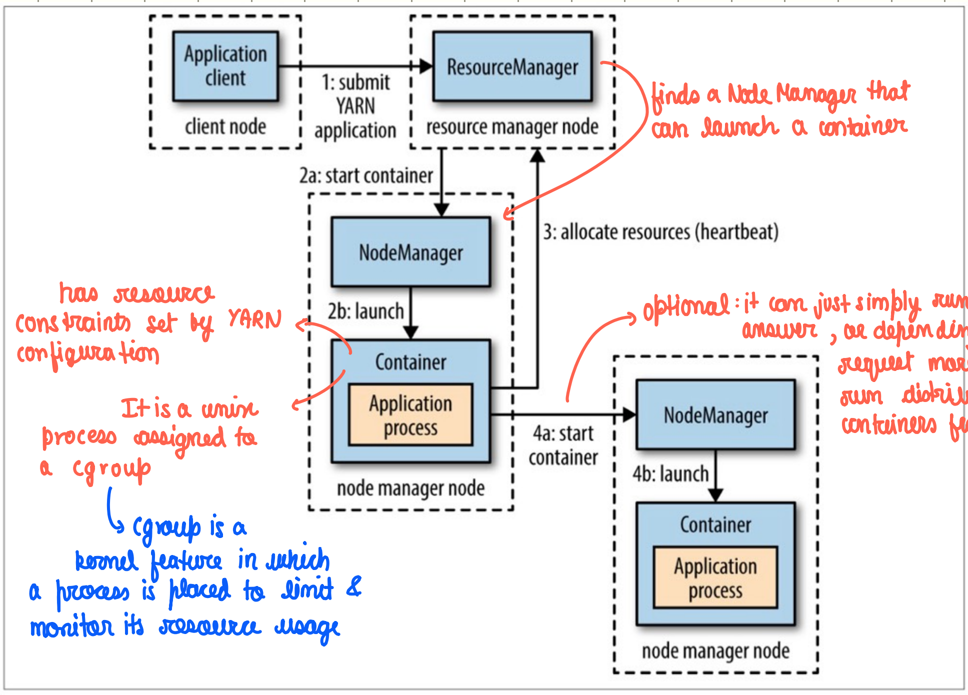


Figure 4-2. How YARN runs an application

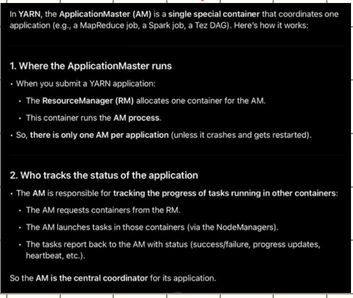
finds a Node Manager that can launch a container

has resource constraints set by YARN configuration

It is a unix process assigned to a cgroup

cgroup is a kernel feature in which a process is placed to limit & monitor its resource usage

optional: it can just simply run computation on one node & return answer, or depending on the application (like MR), it can request more containers from resource manager to run distributed computation. Negotiating more containers from RM is task of application master



In **YARN**, the **ApplicationMaster (AM)** is a **single special container** that coordinates one application (e.g., a MapReduce job, a Spark job, a Tez DAG). Here's how it works:

1. Where the ApplicationMaster runs

- When you submit a YARN application:
  - The **ResourceManager (RM)** allocates one container for the AM.
  - This container runs the **AM process.**
- So, **there is only one AM per application** (unless it crashes and gets restarted).

2. Who tracks the status of the application

- The **AM** is responsible for **tracking the progress of tasks running in other containers:**
  - The AM requests containers from the RM.
  - The AM launches tasks in those containers (via the NodeManagers).
  - The tasks report back to the AM with status (success/failure, progress updates, heartbeat, etc.).

So the **AM is the central coordinator** for its application.

Yarn resource requests

- Has flexible model for making resource requests from Resource Manager

- Application master requests RM to allocate a set of containers & can specify 2 things
  1) Computer resources (CPU, mem) for each container
  2) Locality constraints for the containers

Locality constraints help - ensure the containers use cluster bandwidth efficiently
⇒ would prefer not all containers on the same node to exploit full bandwidth of each node
- ensure containers are allocated close to the data they want to process ⇒ data locality also kept in mind

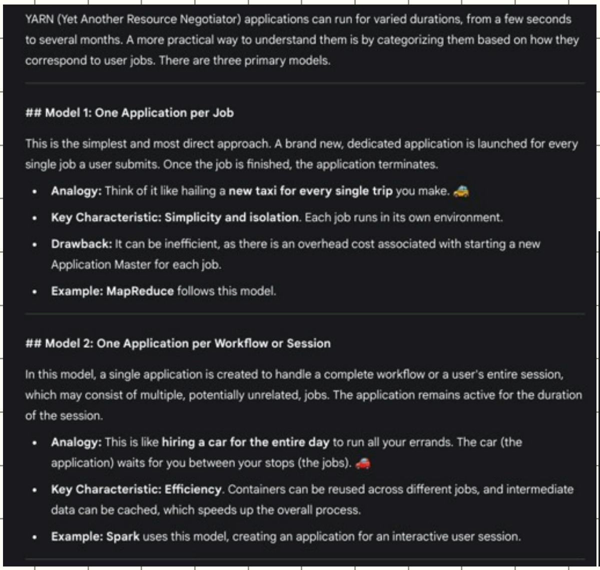
⇒ The request sent by AM to RM contains
1) how much resources for each container,
2) no of containers
3) locality for each container (hostname / rack name)
4) priority of request

locality can be requested on node basis or rack basis
First try allocating on requested node, if cannot, try in same rack & if cannot then relax constraints & allocate somewhere off rack to avoid starvation.

it could be a node holding a replica of the data we need

- The model is flexible, resource requests can be made any time an application is running. AM can request all upfront & also dynamically request.

Yarn application lifespan



YARN (Yet Another Resource Negotiator) applications can run for varied durations, from a few seconds to several months. A more practical way to understand them is by categorizing them based on how they correspond to user jobs. There are three primary models.

Model 1: One Application per Job

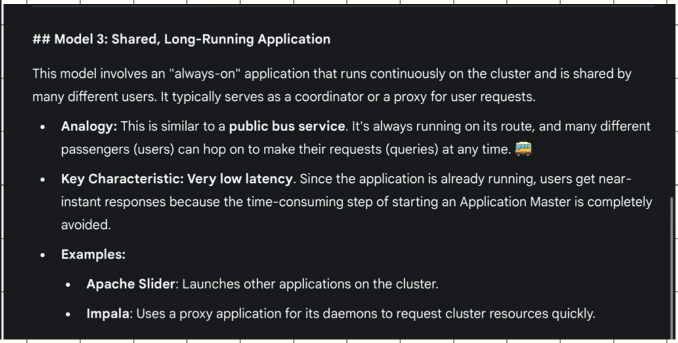
This is the simplest and most direct approach. A brand new, dedicated application is launched for every single job a user submits. Once the job is finished, the application terminates.

- **Analogy:** Think of it like hailing a **new taxi for every single trip** you make. 🚕
- **Key Characteristic: Simplicity and isolation**. Each job runs in its own environment.
- **Drawback:** It can be inefficient, as there is an overhead cost associated with starting a new Application Master for each job.
- **Example: MapReduce** follows this model.

Model 2: One Application per Workflow or Session

In this model, a single application is created to handle a complete workflow or a user's entire session, which may consist of multiple, potentially unrelated, jobs. The application remains active for the duration of the session.

- **Analogy:** This is like **hiring a car for the entire day** to run all your errands. The car (the application) waits for you between your stops (the jobs). 🚗
- **Key Characteristic: Efficiency**. Containers can be reused across different jobs, and intermediate data can be cached, which speeds up the overall process.
- **Example: Spark** uses this model, creating an application for an interactive user session.

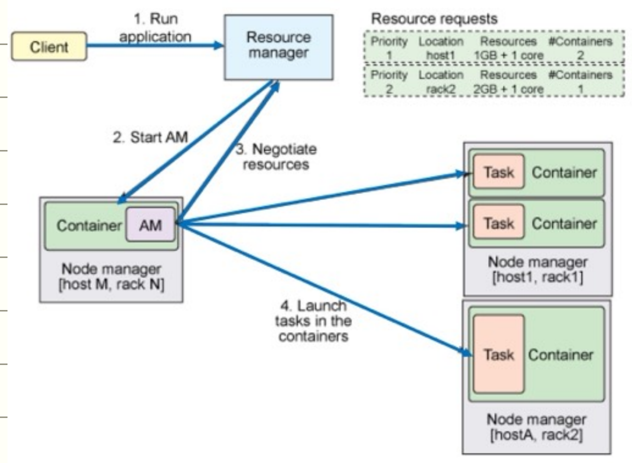


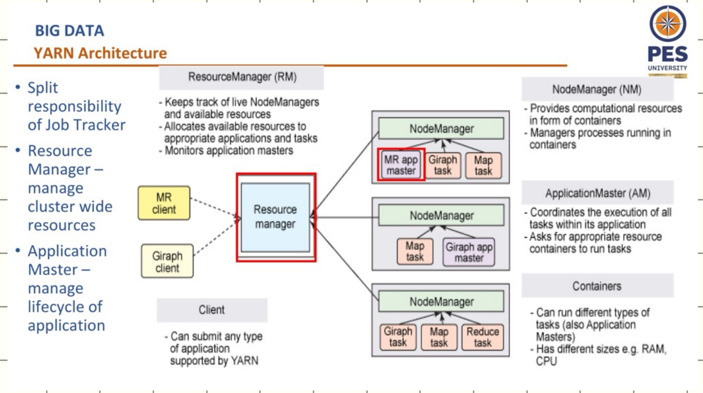
Model 3: Shared, Long-Running Application

This model involves an "always-on" application that runs continuously on the cluster and is shared by many different users. It typically serves as a coordinator or a proxy for user requests.

- **Analogy:** This is similar to a **public bus service**. It's always running on its route, and many different passengers (users) can hop on to make their requests (queries) at any time. 🚌
- **Key Characteristic: Very low latency**. Since the application is already running, users get near-instant responses because the time-consuming step of starting an Application Master is completely avoided.
- **Examples:**
  - **Apache Slider**: Launches other applications on the cluster.
  - **Impala**: Uses a proxy application for its daemons to request cluster resources quickly.

Overview :


Client
1. Run application
Resource manager
Resource requests
Priority Location Resources #Containers
1 host1 1GB + 1 core 2
Priority Location Resources #Containers
2 rack2 2GB + 1 core 1
2. Start AM
3. Negotiate resources
Container AM
Node manager [host M, rack N]
Task Container
Task Container
Node manager [host1, rack1]
4. Launch tasks in the containers
Task Container
Node manager [hostA, rack2]


BIG DATA
YARN Architecture

- Split responsibility of Job Tracker
- Resource Manager – manage cluster wide resources
- Application Master – manage lifecycle of application

ResourceManager (RM)
- Keeps track of live NodeManagers and available resources
- Allocates available resources to appropriate applications and tasks
- Monitors application masters

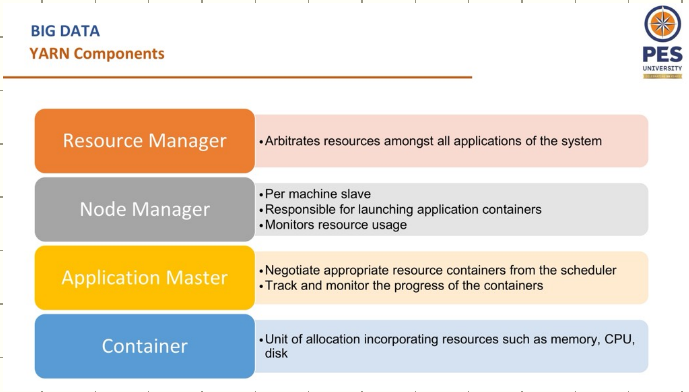
NodeManager (NM)
- Provides computational resources in form of containers
- Managers processes running in containers

ApplicationMaster (AM)
- Coordinates the execution of all tasks within its application
- Asks for appropriate resource containers to run tasks

Containers
- Can run different types of tasks (also Application Masters)
- Has different sizes e.g. RAM, CPU

Client
- Can submit any type of application supported by YARN

PES UNIVERSITY
MR client
Giraph client
Resource manager
NodeManager
MR app master
Giraph task
Map task
NodeManager
Map task
Giraph app master
NodeManager
Giraph task
Map task
Reduce task


BIG DATA
YARN Components

Resource Manager	•Arbitrates resources amongst all applications of the system
Node Manager	•Per machine slave •Responsible for launching application containers •Monitors resource usage
Application Master	•Negotiate appropriate resource containers from the scheduler •Track and monitor the progress of the containers
Container	•Unit of allocation incorporating resources such as memory, CPU, disk

Scheduling in YARN

- Ideally resource requests would be granted immedietly.
- But since resources are limited, YARN will queue them they can be fulfilled.

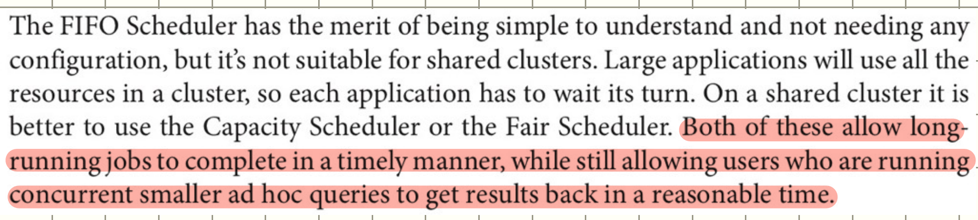
YARN gives us 3 schedulers
1] FIFO
2] Capacity scheduler
3] Fair scheduler

1] FIFO very simple, no configuration needed
↳ Application 1 sends requests ⟶ queue ⟶ All these requests served first
↳ Application 2 sends requests ⟶ queue

⇓

Issue with FIFO
⇒ If App1 large, all its requests will be served first, even if App2 very small, it'll have to wait long

⇒ On shared clusters, its better to use capacity or fair schedulers.



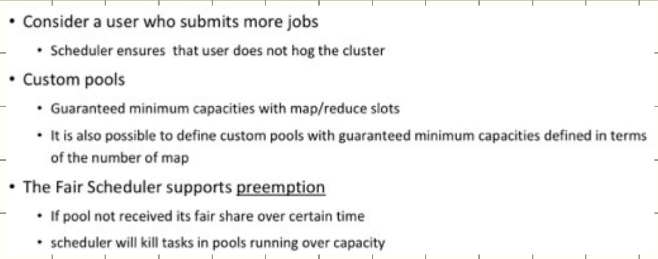
The FIFO Scheduler has the merit of being simple to understand and not needing any configuration, but it's not suitable for shared clusters. Large applications will use all the resources in a cluster, so each application has to wait its turn. On a shared cluster it is better to use the Capacity Scheduler or the Fair Scheduler. Both of these allow long-running jobs 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.

2] Capacity scheduler has a seperate dedicated queue ⇒ small jobs go here & start as soon as they are submitted
↳ one or more

Extra queue does have little overhead ∵ some capacity is kept away for small jobs
⇒ long tasks take a tad bit longer than FIFO
Each queue internally is FIFO scheduled
Issue : if one queue is not being used, no other job can use the extra unused resources

3] Fair Scheduler dynamically balances resources b/w all running jobs.
If only one job present ⇒ it uses 100% resources, if second comes, it is allocated 1/2 resources
⇒ fair sharing

there is lil bit lag before 2nd job starts ∵ first job is freeing up some space for it.



- Consider a user who submits more jobs
  - Scheduler ensures that user does not hog the cluster
- Custom pools
  - Guaranteed minimum capacities with map/reduce slots
  - It is also possible to define custom pools with guaranteed minimum capacities defined in terms of the number of map
- The Fair Scheduler supports preemption
  - If pool not received its fair share over certain time
  - scheduler will kill tasks in pools running over capacity

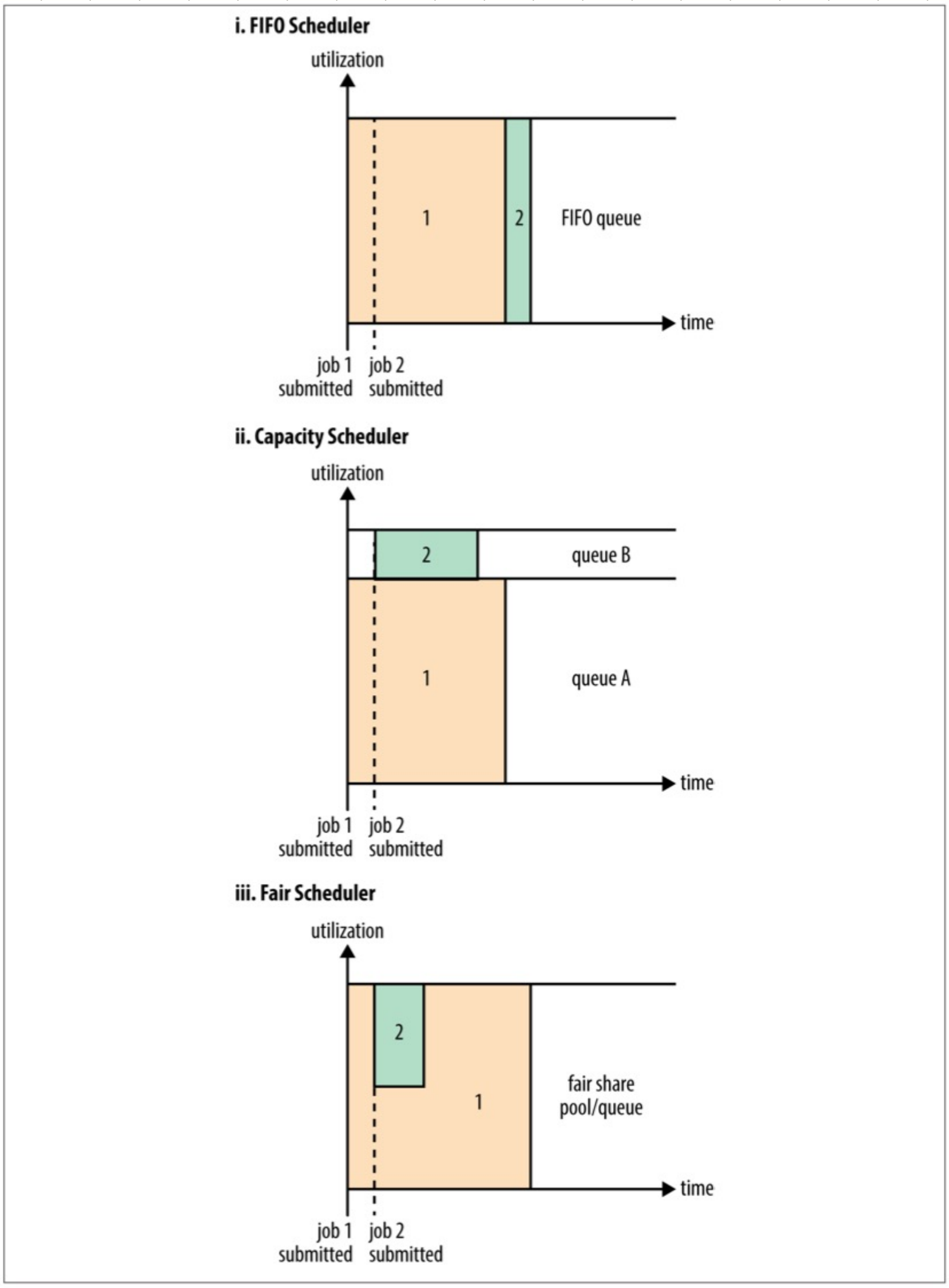
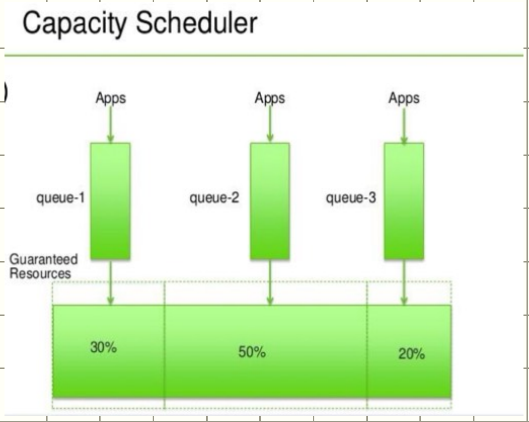
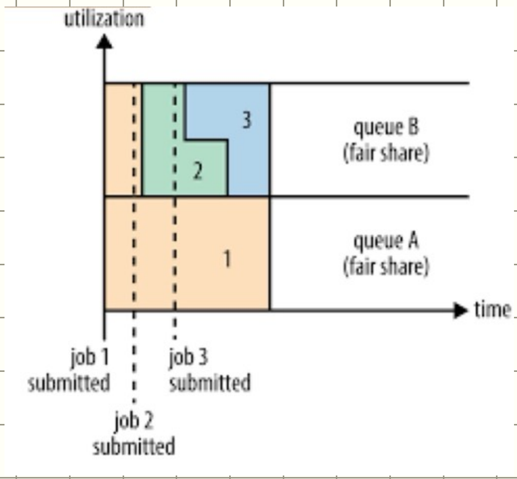


Figure 4-3. Cluster utilization over time when running a large job and a small job under the FIFO Scheduler (i), Capacity Scheduler (ii), and Fair Scheduler (iii)



Fair Scheduler


utilization
3
2
queue B (fair share)
1
queue A (fair share)
time
job 1 submitted
job 3 submitted
job 2 submitted