Managing Resources – Using Schedulers

The YARN Scheduler

- **The ResourceManager's scheduling component (the YARN Scheduler) is responsible for assigning available resources to YARN applications**
  - The scheduler decides *where* and *when* containers will be allocated to applications
  - Based on requirements of each application
  - Containers granted specific resources
    - memory, CPU
- **Administrators define a scheduling policy that best fits requirements**
  - For example, a policy that allocates resources equally among YARN applications
- **The scheduling policy establishes rules for resource sharing**
  - Rules are well-defined
  - Rules form the basis for application start and completion expectations

Scheduler Types

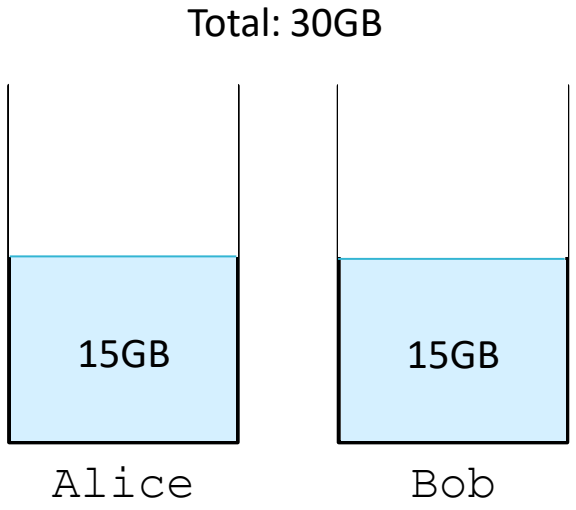
- **YARN supports the following schedulers:**
  - FIFO Scheduler
    - Resources allocated based on time of arrival
  - Capacity Scheduler
    - Resources allocated to pools
    - FIFO scheduling within each pool
  - Fair Scheduler
    - Resources allocated to weighted pools
    - Fair sharing within each pool
- **The Fair Scheduler is the default YARN scheduler in CDH 5**
  - The `yarn.resourcemanager.scheduler.class` property specifies the scheduler in use

The Fair Scheduler

- **The Fair Scheduler is the YARN Scheduler that Cloudera recommends for production clusters**
- **The Fair Scheduler organizes YARN applications into pools**
  - Pools are also known as a *queues* in YARN terminology
  - Each user get a pool named after the user (by default)
  - Resources are divided fairly between the pools (by default)
- **Fair Scheduler characteristics**
  - Allows resources to be controlled proportionally
  - Promotes efficient utilization of cluster resources
  - Promotes fairness between schedulable entities
  - Allows short interactive and long production applications to co-exist
  - Awards resources to pools that are most underserved
  - Gives a container to the pool that has the fewest resources allocated

Fair Scheduler Pools

- **Each application is assigned to a *pool***
- **All pools descend from the `root` pool**
- **Pools can be nested as subpools in which case siblings share the parent's resources**
- **Physical resources are not bound to any specific pool**
- **Pools can be predefined or defined dynamically by specifying a pool name when you submit an application**

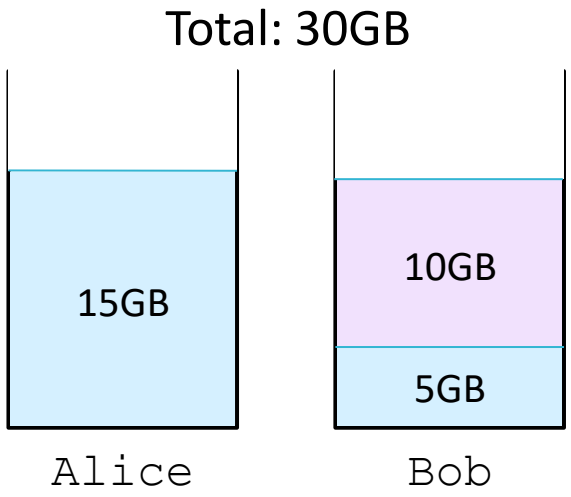


Determining the Fair Share

- **The fair share of resources assigned to the pool is based on**
  - The total resources available across the cluster
  - The number of pools competing for cluster resources
- **Excess cluster capacity is spread across all pools**
  - The aim is to maintain the most even allocation possible so every pool receives its fair share of resources
- **The fair share will never be higher than the actual demand**
- **Pools can use more than their fair share when other pools are not in need of resources**
  - This happens when there are no tasks eligible to run in other pools

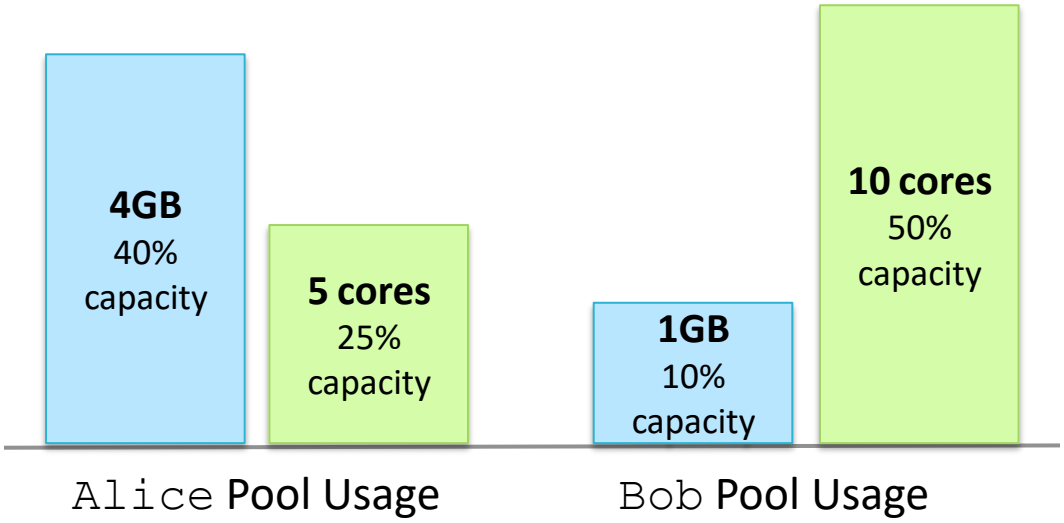
Single Resource Fairness

- **Fair scheduling with Single Resource Fairness**
  - Schedules applications on the basis of a single resource: memory
- **Example**
  - Two pools: `Alice` has been allocated 15GB, and `Bob` has 5GB
  - Both pools request a 10GB container of memory
  - `Bob` has less resources at the moment and will be granted the next 10GB that becomes available



Dominant Resource Fairness

- **Fair scheduling with Dominant Resource Fairness (*recommended*)**
  - Schedules applications on the basis of both memory and CPU
- **Example: A cluster has 10GB of total memory and 20 cores**
  - Pool `Alice` has containers granted for 4GB of memory and 5 cores
  - Pool `Bob` has containers granted for 1GB of memory and 10 cores
  - `Alice` will receive the next container because its 40% dominant share of memory is less than the `Bob` pool's 50% dominant share of CPU

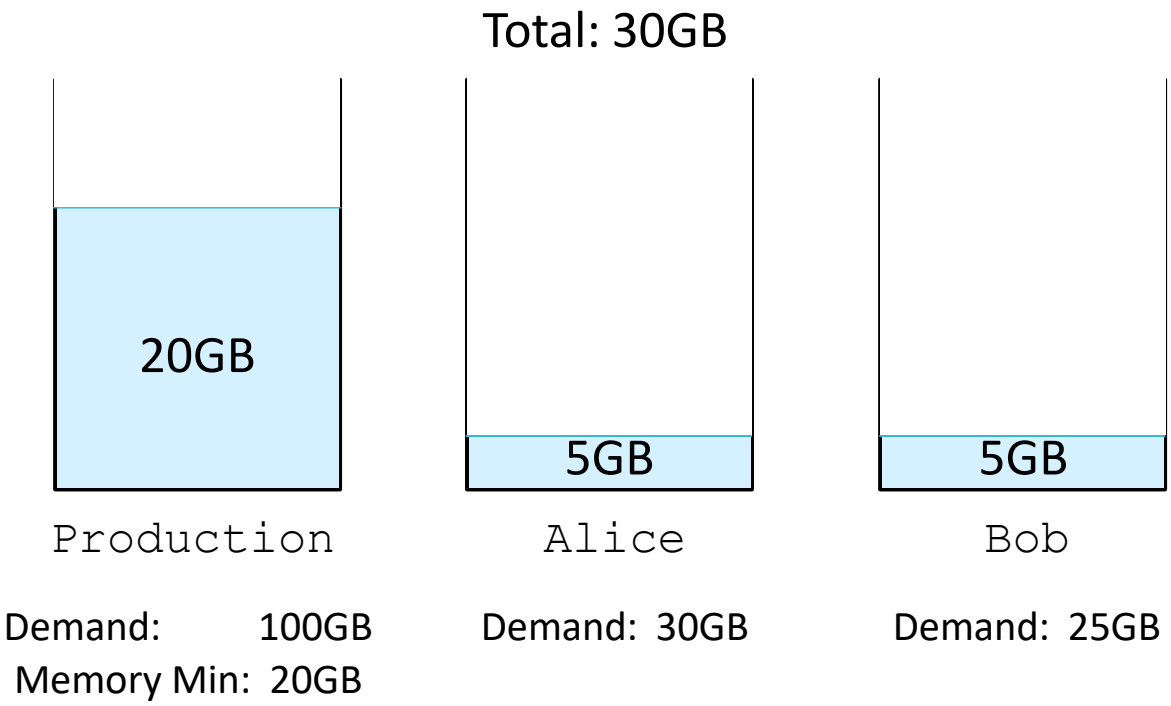


Minimum Resources

- **A pool with minimum resources defined receives priority during resource allocation**
- **The minimum resources are the minimum amount of resources that must be allocated to the pool *prior* to fair share allocation**
  - Minimum resources are allocated to each pool, assuming there is cluster capacity

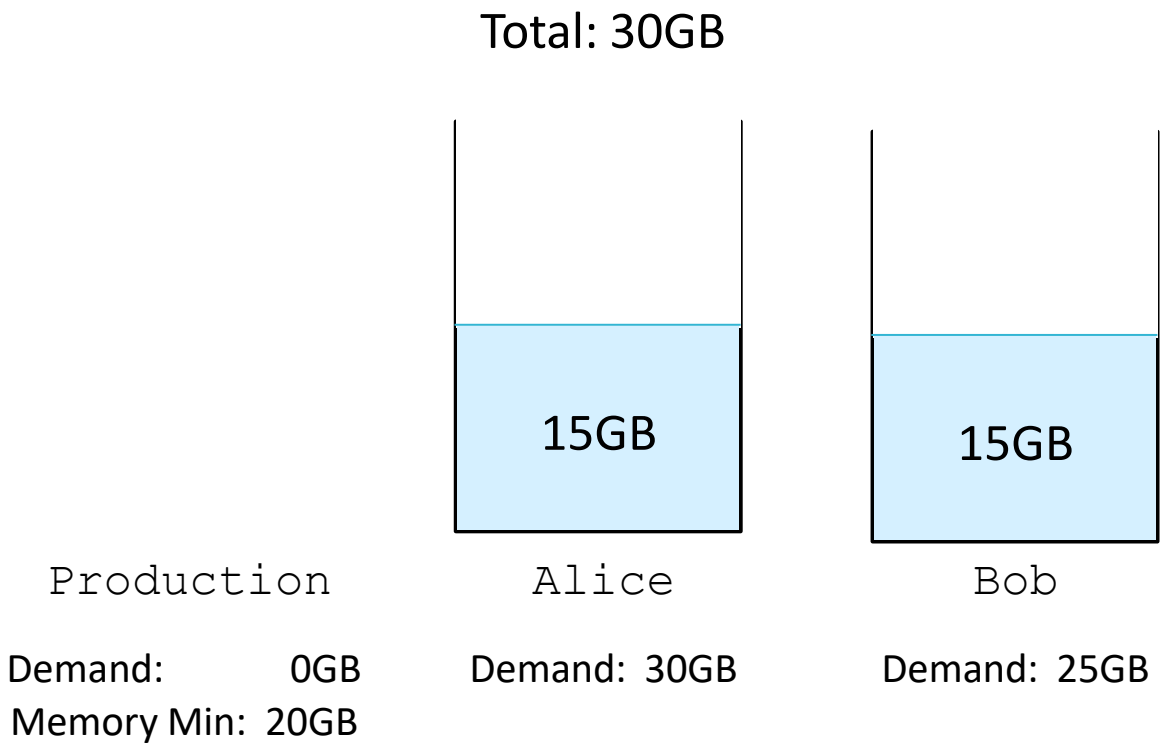
Minimum Allocation Example

- **30GB memory available on the cluster**
  - First, fill up the `Production` pool to the 20GB minimum guarantee
  - Then distribute the remaining 10GB evenly across `Alice` and `Bob`



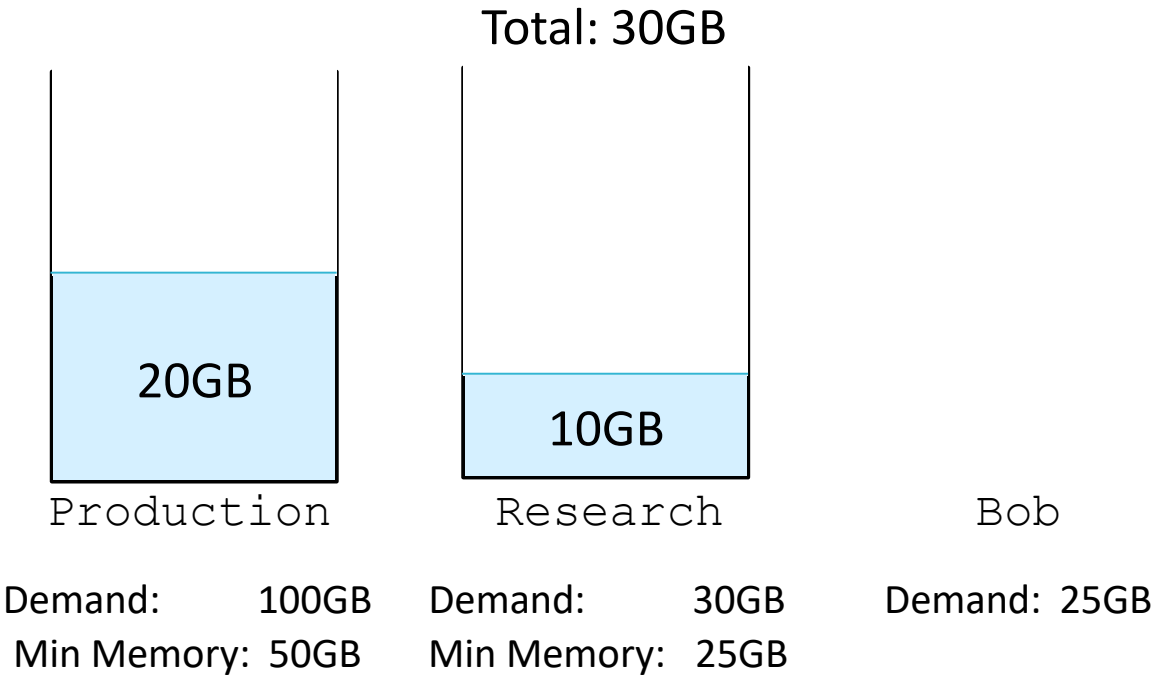
Minimum Allocation Example 2: Production Pool Empty

- **`Production` has no demand, so no resources are allocated to it**
- **All resources are allocated evenly between `Alice` and `Bob`**



Minimum Allocation Example 3: Min Memory Exceeds Resources

- **Combined minimum memory requirements of `Production` and `Research` exceed capacity**
- **Resources are assigned proportionally based on defined minimum resources until available memory is exhausted**
- **No memory remains for pools without min memory defined (i.e., `Bob`)**

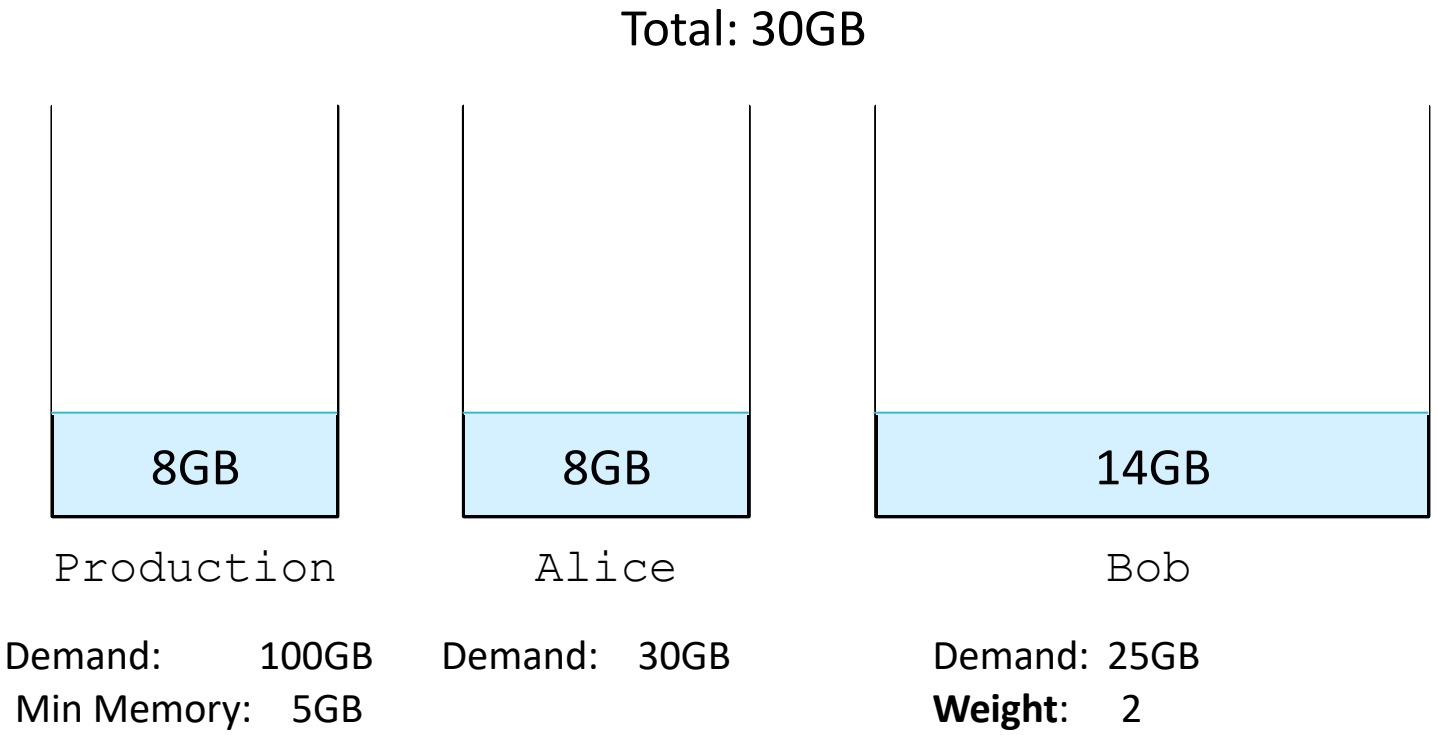


Pools with Weights

- **Instead of (or in addition to) setting minimum resource requirements, pools can be assigned a *weight***
- **Pools with higher weight receive more resources during allocation**
- **'Even water glass height' analogy:**
  - Think of the weight as controlling the 'width' of the glass

Pools with Weights Example: Pool With Double Weight

- **`Production` is filled to minimum memory (5GB)**
- **Remaining 25GB is distributed across pools**
- **`Bob` pool receives twice the amount of memory during fair share allocation**



Fair Scheduler Preemption (1)

- **Recall that pools can use more than their fair share when other pools are not in need of resources**
  - This resource sharing is typically a good thing
- **There may be occasions when a pool needs its fair share back**
  - Example: production applications must run within a set time period
    - Some pools must operate on an SLA (Service Level Agreement)
  - Solution: the preemption feature allows the cluster to be used by less critical applications while also ensuring important applications are not starved for too long
- **If `yarn.scheduler.fair.preemption` is enabled:**
  - the Fair Scheduler kills containers that belong to pools operating over their fair share beyond a configurable timeout
  - Pools operating below fair share receive those reaped resources
- **CM default: preemption is not enabled**

Fair Scheduler Preemption (2)

- **There are two types of preemption available**
  - Minimum share preemption
  - Fair share preemption
- **Preemption avoids killing a container in a pool if it would cause that pool to begin preempting containers in other pools**
  - This prevents a potentially endless cycle of pools killing one another's containers
- **Use fair share preemption conservatively**
  - Set the **Min Share Preemption Timeout** to the number of seconds a pool is under fair share before preemption should begin
  - Default is infinite

Dynamic Resource Pools – Placement Rules (**yarn-site.xml**)

- **configure rules determining in which pool an application will run**
- **`yarn.scheduler.fair.user-as-default-queue`**
  - When true (default), the default pool for an application is *`username`*
  - If false, applications run in the `default` pool
- **`yarn.scheduler.fair.allow-undeclared-pools`**
  - If true (default), applications can declare a new pool at submission
  - If false, applications specifying unknown pools run in `default` pool
- **Specify a pool in MapReduce using `–D mapreduce.job.queuename`**
  - In Spark, use `spark.yarn.queue`
  - In Impala, use `SET REQUEST_POOL=<poolname>`

To use the Fair Scheduler first assign the appropriate scheduler class in yarn-site.xml:

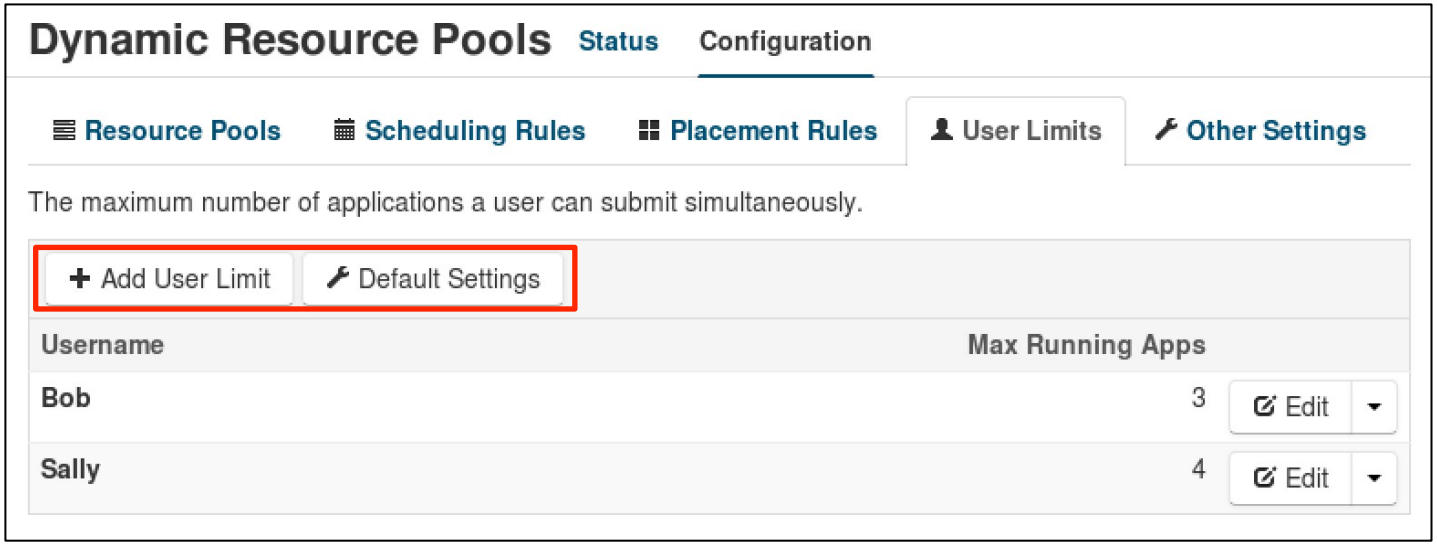
```
<property>
  <name>yarn.resourcemanager.scheduler.class</name>
  <value>org.apache.hadoop.yarn.server.resourcemanager.scheduler.fair.FairScheduler</value>
</property>
```

When Will an Application Run Within a Pool?

- **The Fair Scheduler grants resources to a pool, but which application will get the next resources requested?**
  - It depends on which of the three Fair Scheduler options was chosen for prioritizing applications *within* the pool:
    - Single Resource Fairness
    - Dominant Resource Fairness
    - FIFO
  - It can also depend on whether the Fair Scheduler has been configured to delay assignment of resources when a preferred rack or node is not available
    - This behavior is configured using the `yarn.scheduler.fair.locality.threshold.node` and `yarn.scheduler.fair.locality.threshold.rack` properties

Dynamic Resource Pools – User Limits

- **When defining a pool you can set 'Max Running Apps' for the pool**
- **There is also the option to limit the number of applications specific users can run at the same time across pools**
  - Define a limit for all users in "Default Settings"
  - Define a specific limit for an individual user



Fair Scheduler Settings Deployment

- **These settings are deployed in etc/hadoop/fair-scheduler.xml file on the ResourceManager host**
  - The Fair Scheduler rereads this file every 10 seconds
- **ResourceManager restart is *not* required when the file changes**

```
<allocations>
  <queue name="sales">
    <minResources>10000 mb,0vcores</minResources>
    <maxResources>50000 mb,0vcores</maxResources>
    <weight>2.0</weight>
    <schedulingPolicy>fifo</schedulingPolicy>
    <queue name="emea" />
    <queue name="apac" />
  </queue>
  <queue name="finance">
    <minResources>10000 mb,0vcores</minResources>
    <maxResources>70000 mb,0vcores</maxResources>
    <weight>3.0</weight>
    <schedulingPolicy>fair</schedulingPolicy>
  </queue>
  <queuePlacementPolicy>
    <rule name="specified"   />
    <rule name="primaryGroup" create="false" />
    <rule name="default" queue="finance" />
  </queuePlacementPolicy>
</allocations>
```

Capacity Scheduler Settings Deployment

- The CapacityScheduler is designed to run Hadoop applications as a shared, multi-tenant cluster in an operator-friendly manner while maximizing the throughput and the utilization of the cluster.

- designed to allow sharing a large cluster while giving each organization capacity guarantees.

- 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.

Capacity Scheduler Settings Deployment

- **conf/yarn-site.xml**

yarn.resourcemanager.scheduler.class	org.apache.hadoop.yarn.server.resourcemanager.scheduler.capacity.CapacityScheduler

etc/hadoop/capacity-scheduler.xml is the configuration file for the CapacityScheduler.

```
<property>
  <name>yarn.scheduler.capacity.root.queues</name>
  <value>a,b,c</value>
  <description>The queues at the this level (root is the root queue).
  </description>
</property>

<property>
  <name>yarn.scheduler.capacity.root.a.queues</name>
  <value>a1,a2</value>
  <description>The queues at the this level (root is the root queue).
  </description>
</property>

<property>
  <name>yarn.scheduler.capacity.root.b.queues</name>
  <value>b1,b2,b3</value>
  <description>The queues at the this level (root is the root queue).
  </description>
</property>
```

YARN – Resource Allocation: Worker Node Configuration

`yarn.nodemanager.resource.memory-mb`
Set in YARN / NodeManager Group / Resource Management

- Amount of RAM available on this host for YARN-managed tasks
- Recommendation: the amount of RAM on the host minus the amount needed for non-YARN-managed work (including memory needed by the DataNode daemon)
- Used by the NodeManagers

`yarn.nodemanager.resource.cpu-vcores`
Set in YARN / NodeManager Group / Resource Management

- Number of cores available on this host for YARN-managed tasks
- Recommendation: the number of physical cores on the host minus 1
- Used by the NodeManagers

YARN Scheduler Parameters

`yarn.scheduler.minimum-allocation-mb`
`yarn.scheduler.minimum-allocation-vcores`
Set in YARN / ResourceManager Group / Resource Management

- Minimum amount of memory and cpu cores to allocate for a container
- Task requests lower than these minimums will be set to these values
- CM Defaults: 1 GB, 1 vcore
- Memory recommendation: increase up to 4 GB depending on your developers' requirements
- Cores recommendation: keep the 1 vcore default
- Used by the ResourceManager

`yarn.scheduler.increment-allocation-mb`
`yarn.scheduler.increment-allocation-vcores`
Set in YARN / ResourceManager Group / Resource Management

- Tasks with requests that are not multiples of these increment-allocation values will be rounded up to the nearest increments
- CM Defaults: 512MB and 1 vcore

YARN – Resource Allocation: Memory Request for Containers

`mapreduce.map.memory.mb` and **`mapreduce.reduce.memory.mb`**
Set in YARN / Gateway Group / Resource Management

- Amount of memory to allocate for Map or Reduce tasks
- CM Default: 1 GB
- Recommendation: increase mapreduce.map.memory.mb up to 2 GB, depending on your developers' requirements. Also, set mapreduce.reduce.memory.mb to twice the mapper value.
- Used by clients and NodeManagers

`yarn.app.mapreduce.am.resource.mb`
Set in YARN / Gateway Group / Resource Management

- Amount of memory to allocate for the ApplicationMaster
- CM Default: 1 GB
- Recommendation: 1 GB, however you can increase it if jobs contain many concurrent tasks.
- Used by clients and NodeManagers

YARN – MapReduce Container Heap Size

`yarn.app.mapreduce.am.command-opts`
Set in YARN / Gateway Group

- Java options passed to the ApplicationMaster
- By Default Application Master gets 1GB of heap space
- Used when MapReduce ApplicationMasters are launched

`mapreduce.map.java.opts`
`mapreduce.reduce.java.opts`
Set in YARN / Gateway Group

- Java options passed to Mappers and Reducers
- Default is –Xmx=200m (200MB of heap space)
- Recommendation: increase to a value from 1GB to 4GB, depending on the requirements from your developers
- Used when Mappers and Reducers are launched

YARN – Configure Resource Allocation *and* Process Size Properties

- **The resource allocation properties do *not* determine the heap size for the ApplicationMaster, Mappers, and Reducers**
- **Be sure to adjust both the Java heap size properties *and* the resource allocation properties**
- **For example, if you specified**
  - `yarn.nodemanager.resource.memory-mb` = 8192
  - `mapreduce.map.memory.mb` = 4096
  - And allowed `mapreduce.map.java.opts` to default
- **Then the maximum heap size for Mappers would be 200MB**
  - Because `mapreduce.map.java.opts` defaults to –Xmx=200m
- **Recommendation: set the Java heap size for Mappers and Reducers to 75% of `mapreduce.map.memory.mb` and `mapreduce.reduce.memory.mb`**

Summary of YARN Memory and CPU Default Settings

Service	Setting	Default
YARN Gateway	ApplicationMaster Memory	1GB
YARN Gateway	ApplicationMaster Java Max Heap Size	825MB
YARN Gateway	Map Task Memory	1GB
YARN Gateway	Map Task Max Heap Size	825MB
YARN Gateway	Reduce Task Memory	1GB
YARN Gateway	Reduce Task Max Heap Size	825MB
JobHistory Server	Java Heap Size	1GB
NodeManager	Java Heap Size	1GB
NodeManager	Container Memory	8GB
NodeManager	Container Virtual CPU Cores	8
ResourceManager	Java Heap Size	1GB
ResourceManager	Container Memory Min	1GB
ResourceManager	Container Memory Max	64GB
ResourceManager	Container Virtual CPU Cores Min	1
ResourceManager	Container Virtual CPU Cores Max	32

YARN Tuning Recommendations

- **Inventory the vcores, memory, and disks available on each worker node**
- **Calculate the resources needed for other processes**
  - Reserve 3GB or 20% of total memory for the OS
  - Reserve resources for any non-Hadoop applications
  - Reserve resources for other any Hadoop components
    - HDFS caching (if configured), NodeManager, DataNode
    - Impalad, HBase RegionServer, Solr, etc.
- **Grant the resources not used by the above to your YARN containers**
- **Configure the YARN scheduler and application framework settings**
  - Based on the worker node profile determined above
  - Determine the number of containers needed to best support YARN applications based on the type of workload
  - Monitor usage and tune estimated values to find optimal settings

Hands-On Exercise:

- **Capacity scheduler - 90 Minutes**