Proposal of Guaranteed Capacity Scheduling for YARN

Wei Shao@Rocket Fuel Inc

Last modified: June 19 2015

Introduction

This proposal talks about limitations of the YARN scheduling policies for SLA applications, and tries to solve them by YARN-3806 and the new policy guaranteed capacity scheduling.

Limitations of YARN Fair/Capacity Scheduling

Requirements for batch applications with SLA are:

1. Applications need guaranteed resource share (self contained). If the resource share is unpredictable, the running duration of application is unpredictable. For example, the application may need to know how long it can get 100G memory, and may also want to get promise from scheduler that the resource won't be preempted.
2. Applications with different demand and SLA require different resource share. For example, an application with high resource demand and short deadline should have more resource share than a application with low resource demand and long deadline.

Fair scheduling cannot satisfy these requirements.

1. Application cannot get guaranteed share. Assuming the cluster is fully utilized, when a new application is accepted to the cluster, fair scheduler will try to even the resource allocations of applications, and may even preempt resources of running applications.
2. Applications in one queue cannot always have different resource share since fair scheduling enforces fairness by evening the resources among applications.

Reservation-based capacity scheduling YARN-1051 proposed solution for SLAs by allowing users to reserve capacity over time. However,

1. Application need to predicate resource plan to get a reservation.
2. If prediction is inaccurate, application needs to adapt at runtime to satisfy SLA.
3. It maybe not easy for admin to diagnose scheduling behaviors due to its complexity.

Guaranteed Capacity Scheduling

Guaranteed capacity scheduling makes guarantee to the applications that they can get resources under specified capacity cap in totally predictable manner. The application can meet

SLA more easily since it is self-contained in the shared cluster - external uncertainties are eliminated.

For example, suppose queue A has initial capacity 100G memory, and there are two pending applications 1 and 2, 1's specified capacity is 70G, 2's specified capacity is 50G. Queue A may accept application 1 to run first and makes guarantee that 1 can get resources exponentially up to its capacity and won't be preempted (if allocation of 1 is 5G in scheduling cycle N, demand is 80G, exponential factor is 2. In N+1, it can get 5G, in N+2, it can get 10G, in N+3, it can get 20G, and in N+4, it can get 30G, reach its capacity). Later, when the cluster is free, queue A may decide to scale up by increasing its capacity to 120G, so it can accept application 2 and make guarantee to it as well. Queue A can scale down to its initial capacity when any application completes.

Guaranteed capacity scheduling also has other features the example doesn't illustrate. See details below.

YARN-3806 proposed a generic interface that particular scheduling policy should implement. To introduce guaranteed share scheduling, we just talk about its own scheduling configurations and the implementation of generic interface.

(Note: please read YARN-3806 before continuing, guaranteed capacity scheduling is based on generic scheduling framework proposed there.)

Scheduling Configurations: There are two configuration items for each sub queue.

1. capacity: The maximal resources sub queue can get. The capacity can be static or dynamic. If it is dynamic, it can increase when the parent queue scale up. But it won't be below its initial value in any case.

2. exponentialFactor: The scheduler guarantees that the sub queue's allocation can increase to allocation*exponentialFactor in each scheduling cycle until min(demand, capacity) is reached. Exponential growth helps for the stability of the cluster since each application starts slowly. It is also effective, since bigger application can get resources faster.

Implementation of Generic Scheduling Interface: We only talk about implementations of non trivial functions.

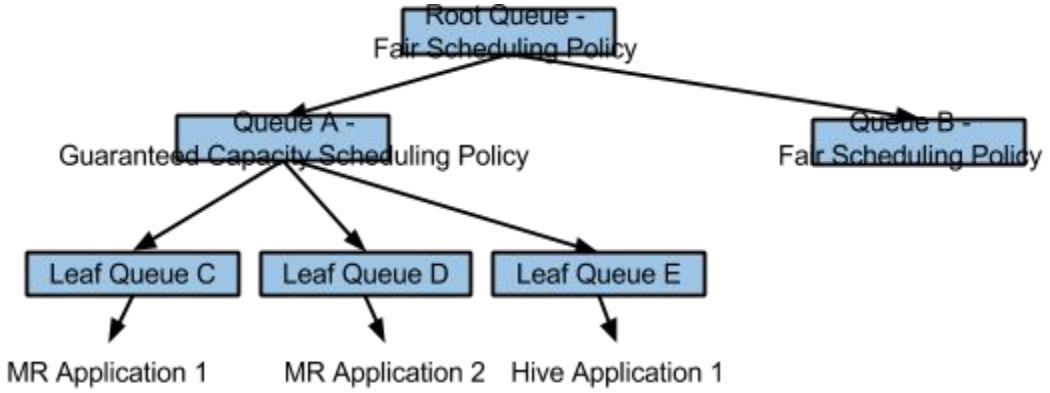

Figure 1

1. dispatchApplications(). In figure 1, suppose hive application 1 is in pending application list of Queue A. dispatchApplications() accepts it and creates a new leaf queue E for it.

2. accommodateConfigurations(). In figure 1, if minimalShare of queue A is static, accommodateConfigurations() of queue A will enforce following invariant:

$$configurations.minimalShare(static) \geq \sum_{child} child.configurations.capacity$$

If invariant is not satisfied, execution will be cancelled (newly accepted application may be rejected). If minimalShare of queue A is dynamic, queue A can scale up/down by increasing/decreasing minimalShare. It can scale up if

$$\begin{aligned} &configurations.minimalShare \\ &< \sum_{child\ with\ static\ capacity} child.configurations.capacity \\ &+ \sum_{child\ with\ dynamic\ capacity} \max(child.configurations.capacity.initialValue, child.demand) \\ &\wedge (resourceManager.clusterResources > rootQueue.demand \vee configurations.minimalShare \\ &< configurations.minimalShare.initialValue) \end{aligned}$$

and scale down if

$$\begin{aligned} &configurations.minimalShare \\ &> \sum_{child\ with\ static\ capacity} child.configurations.capacity \\ &+ \sum_{child\ with\ dynamic\ capacity} \max(child.configurations.capacity.initialValue, child.demand) \\ &\wedge (resourceManager.clusterResources < rootQueue.demand \vee configurations.minimalShare \\ &> configurations.minimalShare.initialValue) \end{aligned}$$

The general ideas for scale up/down are:

   a. When demand is high, the queue can scale up to accept more applications or increase capacities of running applications while still keep promise (exponential resource growth) to all running applications. The queue should be conservative to perform scale up since it needs to hold increased capacity for a while for keeping promise to running applications, capacities for other queues will be suppressed for some time.

b. When demand is low, the queue can scale down each time after running application is completed to allow other queues (especially the queues also use guaranteed capacity) to scale up. The queue should be conservative to perform scale down since it may not be able to get its original capacity back in short time.

c. According to the points above, scale up/down can be harmful when the cluster has more than one queue using guaranteed capacity share, they may compete with each other. So for the queue using guaranteed capacity, initial value of minimalShare in configuration file should just reflect average demand, the queue should always try to keep high water mark.

3. adjustConfigurations(). In figure 1, if minimalShare of queue A is dynamic, adjustConfigurations() of root queue policy may decrease it to enforce invariant

$$resourceManager.clusterResources \geq \sum_{child} child.configurations.minimalShare$$

adjustConfigurations() of queue B will enforce invariant

$$configurations.minimalShare(final) \geq \sum_{child} child.configurations.capacity$$

by adjusting dynamic capacities of sub queues, and ensure

$$\forall\ child\ with\ dynamic\ capacity,$$
$$child.configurations.capacity \geq \max(child.configurations.capacity.initialValue, child.demand)$$

If any invariant above cannot be satisfied, execution will be cancelled.

4. updateCapacities(). The preemption demand for each sub queue is:

$$preemptionDemand = \max(0, \min(allocation * exponentialFactor, capacity, demand) - allocation)$$

The scheduler should guarantee that all preemption demands can be satisfied and no sub queues will be preempted. Additional resources can be distributed to sub queues proportional to their capacity configurations.

Proposed Cluster Configurations

See figure 2 for an example of cluster configurations.

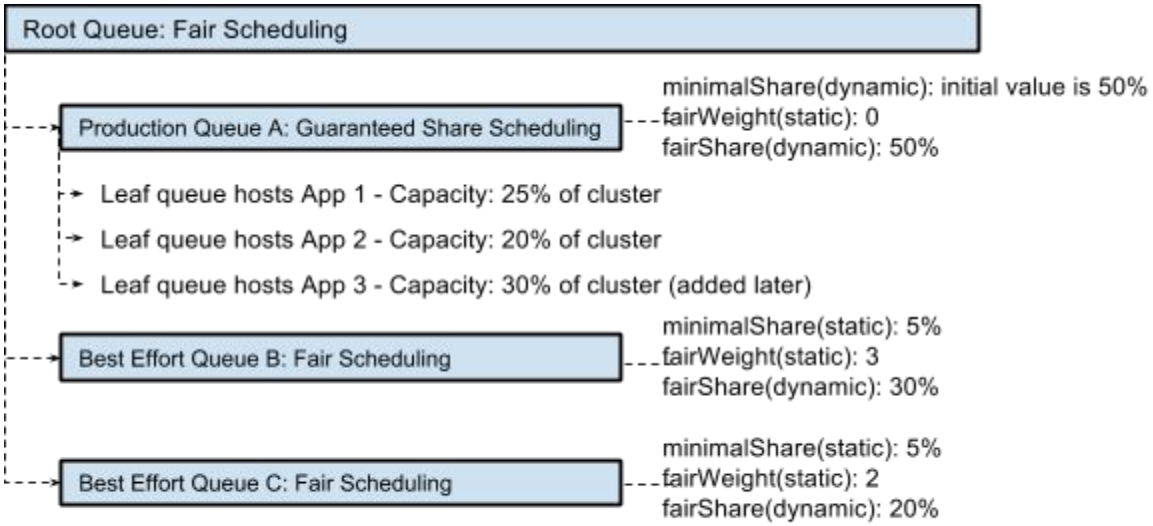

Figure 2

Suppose root queue use fair scheduling, the guidelines for configurations of sub queues are:

1. For production queues using guaranteed share scheduling:
    a. Initial value of minimalShare in configuration file should just reflect average aggregate capacity of running applications. It is usually a large value.
    b. MinimalShare should be dynamic, so production queue can scale up/down.
    c. FairWeight should be set to 0, so minimalShare is the dominant factor for fair scheduler of the root queue to calculate fairShare.
    d. Elastic applications(can progress faster with more resources) should set capacity to dynamic.

2. For best effort queues using other scheduling policies:
    a. MinimalShare should be small, so production queues can scale up.
    b. FairWeight should be positive and it is the dominant factor for fair scheduler of root queue to calculate fairShare.

The changes of dynamic configurations when queue 1 scale up and down is in table below:

States of Queue A and Cluster	**Dynamic Configurations of Queue A**	**Dynamic Configurations of Queue B**	**Dynamic Configurations of Queue C**
Queue A has running apps 1 & 2. The cluster is free.	minimalShare: 50% fairShare: 50%	fairShare: 30%	fairShare: 20%
Queue A has running apps 1, 2 & 3.	minimalShare: 75% fairShare: 75%	fairShare: 15%	fairShare: 10%

The cluster is free.			
Queue A has running apps 1. The cluster is busy.	minimalShare: 25% fairShare: 25%	fairShare: 45%	fairShare: 30%

Discussions

Satisfying SLAs and keeping cluster utilized effectively are usually two conflicting goals. Every scheduling policies have to make tradeoffs. Pros and cons of guaranteed capacity scheduling and comparison with reservation based scheduling are discussed in this section.

Pros:

1. Application is self-contained after it is accepted to run. It knows the maximal resources it can get, and can get resources in totally predictable manner (exponential growth). In reservation based scheduling, scheduling planner decides predicated duration/complete time for the application according to all current reservations, and the application may be converted to the best effort job if resource reserved is not enough. So guaranteed capacity scheduling provide better self-containess for the application, the promise scheduler made to application is also easier to understand. The application can predict maximal running to completion duration based on historical executions easily.

2. Scheduling behavior is easy to understand, debug and diagnose. Admins can monitor/manage running & pending applications efficiently. In reservation based scheduling, admins may need more skills and efforts.

Cons:

1. For applications with rigid and rough demand skyline, reservation based scheduler can work more effectively since it can pack workflows of different applications to best utilize cluster. By guaranteed capacity scheduling, allocation of such applications may below its capacity for most time. As a result, cluster could be underutilized if applications in best effects queues cannot fill the gaps efficiently. However, if application master has its own task scheduler/resource pool mechanisms to utilize available resources elastically, this won't be a big issue. Currently, MR application master has its own scheduler. Spark, Tez (Hive and Pig can run on it), Oozie already support or propose such mechanisms. See figure below for a possible execution timeline of MR application.

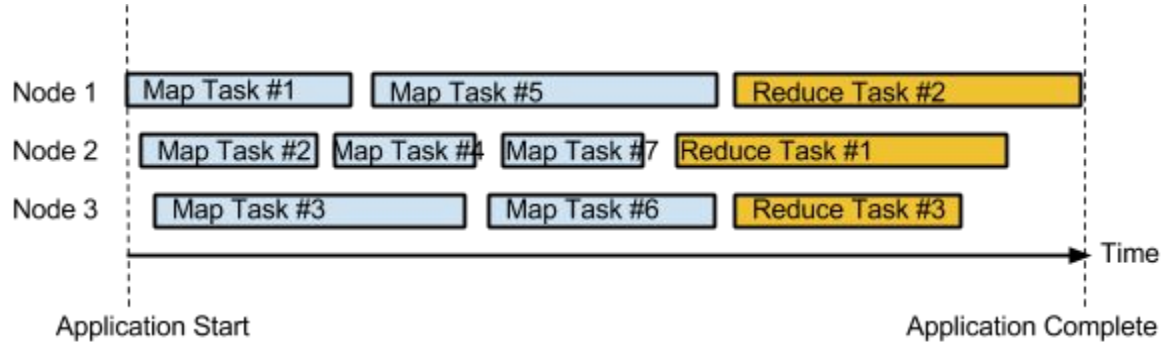

Figure 3

2. Reservation based scheduler provide better latency for best effort applications. However, in large cluster with lots of SLA applications, aggregate allocation skyline of all SLA applications may be evenly enough to allow best effort applications to run at any time.

Future Work

1. Develop tools to find best capacity config for batch applications. For example, running application with different capacity configs to find sweet point by which the application can run most effectively. In figure 4, the point P which has minimal rectangle area is the sweet point.

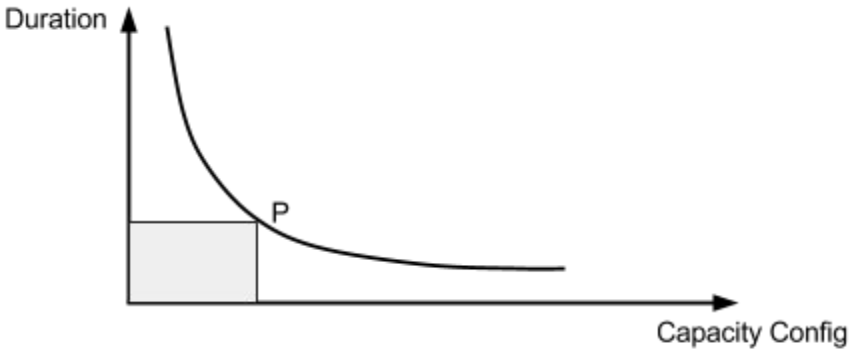

Figure 4

2. The demand function for the batch application with decent task scheduler should like the one in figure 5. In initialization phase, application only acquires resources, in working phase, application may both acquires and releases resources, and in end phase, application only release resources. So to optimize overall resource utilization, the application can send event to scheduler when it enters end phase. The scheduler can then downgrade the guarantee to application from ‘exponential resource growth and no preemption’ to only ‘no preemption’.

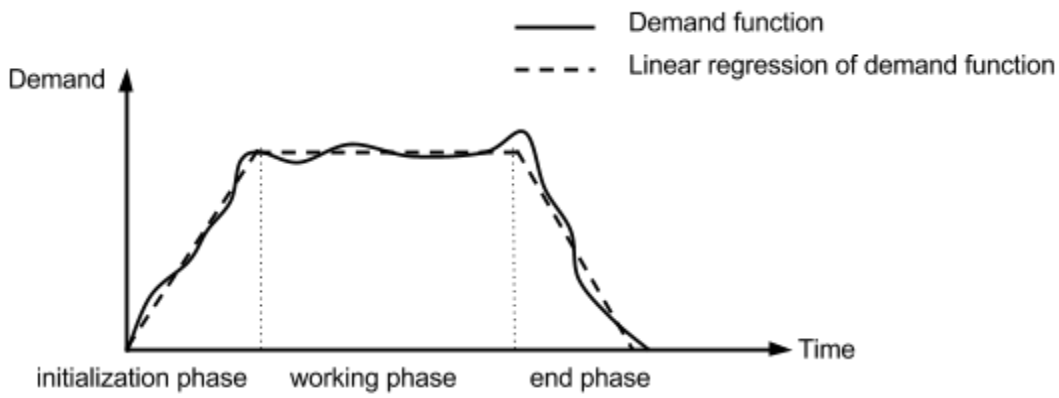

Figure 5

3. Design ordering policy for pending applications. Applications can be ordered according to the deadline and expected execution time based on historical executions.

4. Write prototype to evaluate proposed scheduling policy.

Related Efforts

YARN-3806 Proposal of Generic Scheduling Framework for YARN
YARN-3808 Proposal of Time Extended Fair Scheduling for YARN
YARN-1051 Reservation Based Scheduling