

Hadoop Fair Scheduler Design Document

October 18, 2010

Contents

1	Introduction	2
2	Fair Scheduler Goals	2
3	Scheduler Features	2
3.1	Pools	2
3.2	Minimum Shares	3
3.3	Preemption	3
3.4	Running Job Limits	4
3.5	Job Priorities	4
3.6	Pool Weights	4
3.7	Delay Scheduling	4
3.8	Administration	4
4	Implementation	5
4.1	Hadoop Scheduling Background	5
4.2	Fair Scheduler Basics	6
4.3	The <code>Schedulable</code> Class	6
4.4	Fair Sharing Algorithm	7
4.5	Preemption	7
4.6	Fair Share Computation	7
4.7	Running Job Limits	8
4.8	Delay Scheduling	9
4.9	Locking Order	10
4.10	Unit Tests	10
5	Code Guide	11

1 Introduction

The Hadoop Fair Scheduler started as a simple means to share MapReduce clusters. Over time, it has grown in functionality to support hierarchical scheduling, preemption, and multiple ways of organizing and weighing jobs. This document explains the goals and features of the Fair Scheduler and its internal design.

2 Fair Scheduler Goals

The Fair Scheduler was designed with four main goals:

1. Run small jobs quickly even if they are sharing a cluster with large jobs. Unlike Hadoop’s built-in FIFO scheduler, fair scheduling lets small jobs make progress even if a large job is running, without starving the large job.
2. Provide guaranteed service levels to “production” jobs, to let them run alongside experimental jobs in a shared cluster.
3. Be simple to administer and configure. The scheduler should do something reasonable “out of the box,” and users should only need to configure it as they discover that they want to use more advanced features.
4. Support reconfiguration at runtime, without requiring a cluster restart.

3 Scheduler Features

This section provides a quick overview of the features of the Fair Scheduler. A detailed usage guide is available in the Hadoop documentation in `build/docs/fair_scheduler.html`.

3.1 Pools

The Fair Scheduler groups jobs into “pools” and performs fair sharing between these pools. Each pool can use either FIFO or fair sharing to schedule jobs internal to the pool. The pool that a job is placed in is determined by a JobConf property, the “pool name property”. By default, this is `mapreduce.job.user.name`, so that there is one pool per user. However, different properties can be used, e.g. `group.name` to have one pool per Unix group.

A common trick is to set the pool name property to an unused property name such as `pool.name` and make this default to `mapreduce.job.user.name`, so that there is one pool per user but it is also possible to place jobs into “special” pools by setting their `pool.name` directly. The `mapred-site.xml` snippet below shows how to do this:

```
<property>
  <name>mapred.fairscheduler.poolnameproperty</name>
  <value>pool.name</value>
</property>
```

```
<property>
  <name>pool.name</name>
  <value>${mapreduce.job.user.name}</value>
</property>
```

3.2 Minimum Shares

Normally, active pools (those that contain jobs) will get equal shares of the map and reduce task slots in the cluster. However, it is also possible to set a *minimum share* of map and reduce slots on a given pool, which is a number of slots that it will always get when it is active, even if its fair share would be below this number. This is useful for guaranteeing that production jobs get a certain desired level of service when sharing a cluster with non-production jobs. Minimum shares have three effects:

1. The pool's fair share will always be at least as large as its minimum share. Slots are taken from the share of other pools to achieve this. The only exception is if the minimum shares of the active pools add up to more than the total number of slots in the cluster; in this case, each pool's share will be scaled down proportionally.
2. Pools whose running task count is below their minimum share get assigned slots first when slots are available.
3. It is possible to set a *preemption timeout* on the pool after which, if it has not received enough task slots to meet its minimum share, it is allowed to kill tasks in other jobs to meet its share. Minimum shares with preemption timeouts thus act like SLAs.

Note that when a pool is inactive (contains no jobs), its minimum share is not "reserved" for it – the slots are split up among the other pools.

3.3 Preemption

As explained above, the scheduler may kill tasks from a job in one pool in order to meet the minimum share of another pool. We call this preemption, although this usage of the word is somewhat strange given the normal definition of preemption as pausing; really it is the *job* that gets preempted, while the task gets killed. The feature explained above is called *min share preemption*. In addition, the scheduler supports *fair share preemption*, to kill tasks when a pool's fair share is not being met. Fair share preemption is much more conservative than min share preemption, because pools without min shares are expected

to be non-production jobs where some amount of unfairness is tolerable. In particular, fair share preemption activates if a pool has been below *half* of its fair share for a configurable fair share preemption timeout, which is recommended to be set fairly high (e.g. 10 minutes).

In both types of preemption, the scheduler kills the most recently launched tasks from over-scheduled pools, to minimize the amount of computation wasted by preemption.

3.4 Running Job Limits

The fair scheduler can limit the number of concurrently running jobs from each user and from each pool. This is useful for limiting the amount of intermediate data generated on the cluster. The jobs that will run are chosen in order of submit time and priority. Jobs submitted beyond the limit wait for one of the running jobs to finish.

3.5 Job Priorities

Within a pool, job priorities can be used to control the scheduling of jobs, whether the pool's internal scheduling mode is FIFO or fair sharing:

- In FIFO pools, jobs are ordered first by priority and then by submit time, as in Hadoop's default scheduler.
- In fair sharing pools, job priorities are used as weights to control how much share a job gets. The normal priority corresponds to a weight of 1.0, and each level gives 2x more weight. For example, a high-priority job gets a weight of 2.0, and will therefore get 2x the share of a normal-priority job.

3.6 Pool Weights

Pools can be given weights to achieve unequal sharing of the cluster. For example, a pool with weight 2.0 gets 2x the share of a pool with weight 1.0.

3.7 Delay Scheduling

The Fair Scheduler contains an algorithm called delay scheduling to improve data locality. Jobs that cannot launch a data-local map task wait for some period of time before they are allowed to launch non-data-local tasks, ensuring that they will run locally if some node in the cluster has the relevant data. Delay scheduling is described in detail in Section 4.8.

3.8 Administration

The Fair Scheduler includes a web UI displaying the active pools and jobs and their fair shares, moving jobs between pools, and changing job priorities. In addition, the Fair Scheduler's allocation file (specifying min shares and preemption timeouts for the pools) is automatically reloaded if it is modified on disk, to allow runtime reconfiguration.

4 Implementation

4.1 Hadoop Scheduling Background

Hadoop jobs consist of a number of map and reduce *tasks*. These tasks run in *slots* on the nodes on the cluster. Each node is configured with a number of map slots and reduce slots based on its computational resources (typically one slot per core). The role of the scheduler is to assign tasks to any slots that are free.

All schedulers in Hadoop, including the Fair Scheduler, inherit from the `TaskScheduler` abstract class. This class provides access to a `TaskTrackerManager` – an interface to the JobTracker – as well as a `Configuration` instance. It also asks the scheduler to implement three abstract methods: the lifecycle methods `start` and `terminate`, and a method called `assignTasks` to launch tasks on a given TaskTracker. Task assignment in Hadoop is reactive. TaskTrackers periodically send heartbeats to the JobTracker with their `TaskTrackerStatus`, which contains a list of running tasks, the number of slots on the node, and other information. The JobTracker then calls `assignTasks` on the scheduler to obtain tasks to launch. These are returned with the heartbeat response.

Apart from reacting to heartbeats through `assignTasks`, schedulers can also be notified when jobs have been submitted to the cluster, killed, or removed by adding listeners to the `TaskTrackerManager`. The Fair Scheduler sets up these listeners in its `start` method. An important role of the listeners is to initialize jobs that are submitted – until a job is initialized, it cannot launch tasks. The Fair Scheduler currently initializes all jobs right away, but it may also be desirable to hold off initializing jobs if too many are submitted to limit memory usage on the JobTracker.

Selection of tasks *within* a job is mostly done by the `JobInProgress` class, and not by individual schedulers. `JobInProgress` exposes two methods, `obtainNewMapTask` and `obtainNewReduceTask`, to launch a task of either type. Both methods may either return a `Task` object or `null` if the job does not wish to launch a task. Whether a job wishes to launch a task may change back and forth during its lifetime. Even after all tasks in the job have been started, the job may wish to run another task for speculative execution. In addition, if the node containing a map task failed, the job will wish to re-run it to rebuild its output for use in the reduce tasks. Schedulers may therefore need to poll multiple jobs until they find one with a task to run.

Finally, for map tasks, an important scheduling criterion is data locality: running the task on a node or rack that contains its input data. Normally, `JobInProgress.obtainNewMapTask` returns the “closest” map task to a given node. However, to give schedulers slightly more control over data locality, there is also a version of `obtainNewMapTask` that allows the scheduler to cap the level of non-locality allowed for the task (e.g. request a task only on the same node, or `null` if none is available). The Fair Scheduler uses this method with an algorithm called delay scheduling (Section 4.8) to optimize data locality.

4.2 Fair Scheduler Basics

At a high level, the Fair Scheduler uses hierarchical scheduling to assign tasks. First it selects a pool to assign a task to according to the fair sharing algorithm in Section 4.4. Then it asks the pool obtain a task. The pool chooses among its jobs according to its internal scheduling order (FIFO or fair sharing).

In fact, because jobs might not have tasks to launch (`obtainNew(Map|Reduce)Task` can return null), the scheduler actually establishes an ordering on jobs and asks them for tasks in turn. Within a pool, jobs are sorted either by priority and start time (for FIFO) or by distance below fair share. If the first job in the ordering does not have a task to launch, the pool will ask the second, third, etc jobs. Pools themselves are sorted by distance below min share and fair share, so if the first pool does not have any jobs that can launch tasks, the second pool is asked, etc. This makes it straightforward to implement features like delay scheduling (Section 4.8) that may cause jobs to “pass” on a slot.

Apart from the assign tasks code path, the Fair Scheduler also has a periodic update thread that calls `update` every few seconds. This thread is responsible for recomputing fair shares to display them on the UI (Section 4.6), checking whether jobs need to be preempted (Section 4.5), and checking whether the allocations file has changed to reload pool allocations (through `PoolManager`).

4.3 The Schedulable Class

To allow the same fair sharing algorithm to be used both between pools and within a pool, the Fair Scheduler uses an abstract class called `Schedulable` to represent both pools and jobs. Its subclasses for these roles are `PoolSchedulable` and `JobSchedulable`. A `Schedulable` is responsible for three roles:

1. It can be asked to obtain a task through `assignTask`. This may return `null` if the `Schedulable` has no tasks to launch.
2. It can be queried for information about the pool/job to use in scheduling, such as:
 - Number of running tasks.
 - Demand (number of tasks the `Schedulable` *wants* to run; this is equal to number of running tasks + number of unlaunched tasks).
 - Min share assigned through config file.
 - Weight (for fair sharing).
 - Priority and start time (for FIFO scheduling).
3. It can be assigned a fair share through `setFairShare`.

There are separate `Schedulables` for map and reduce tasks, to make it possible to use the same algorithm on both types of tasks.

4.4 Fair Sharing Algorithm

A simple way to achieve fair sharing is the following: whenever a slot is available, assign it to the pool that has the fewest running tasks. This will ensure that all pool get an equal number of slots, unless a pool's demand is less than its fair share, in which case the extra slots are divided evenly among the other pools. Two features of the Fair Scheduler complicate this algorithm a little:

- Pool weights mean that some pools should get more slots than others. For example, a pool with weight 2 should get 2x more slots than a pool with weight 1. This is accomplished by changing the scheduling rule to “assign the slot to the pool whose value of $runningTasks/weight$ is smallest.”
- Minimum shares mean that pools below their min share should get slots first. When we sort pools to choose which ones to schedule next, we place pools below their min share ahead of pools above their min share. We order the pools below their min share by how far they are below it as a percentage of the share.

This fair sharing algorithm is implemented in `FairShareComparator` in the `SchedulingAlgorithms` class. The comparator orders jobs by distance below min share and then by $runningTasks/weight$.

4.5 Preemption

To determine when to preempt tasks, the Fair Schedulers maintains two values for each `PoolSchedulable`: the last time when the pool was at its min share, and the last time when the pool was at half its fair share. These conditions are checked periodically by the update thread in `FairScheduler.updatePreemptionVariables`, using the methods `isStarvedForMinShare` and `isStarvedForFairShare`. These methods also take into account the demand of the pool, so that a pool is not counted as starving if its demand is below its min/fair share but is otherwise met.

When preempting tasks, the scheduler kills the most recently launched tasks from over-scheduled pools. This minimizes the amount of computation wasted by preemption and ensures that all jobs can eventually finish (it is as if the preempted jobs just never got their last few slots). The tasks are chosen and preempted in `preemptTasks`.

Note that for min share preemption, it is clear when a pool is below its min share because the min share is given as a number of slots, but for fair share preemption, we must be able to compute a pool's fair share to determine when it is being starved. This computation is trickier than dividing the number of slots by the number of pools due to weights, min shares and demands. Section 4.6 explains how fair shares are computed.

4.6 Fair Share Computation

The scheduling algorithm in Section 4.4 achieves fair shares without actually needing to compute pools' numerical shares beforehand. However, for preemption and for displaying

shares in the Web UI, we want to know what a pool’s fair share is even if the pool is not currently at its share. That is, we want to know how many slots the pool *would* get if we started with all slots being empty and ran the algorithm in Section 4.4 until we filled them. One way to compute these shares would be to simulate starting out with empty slots and calling `assignTasks` repeatedly until they filled, but this is expensive, because each scheduling decision takes $O(\text{numJobs})$ time and we need to make $O(\text{numSlots})$ decisions.

To compute fair shares efficiently, the Fair Scheduler includes an algorithm based on binary search in `SchedulingAlgorithms.computeFairShares`. This algorithm is based on the following observation. If all slots had been assigned according to weighted fair sharing respecting pools’ demands and min shares, then there would exist a ratio r such that:

1. Pools whose demand d_i is less than rw_i (where w_i is the weight of the pool) are assigned d_i slots.
2. Pools whose min share m_i is more than rw_i are assigned $\min(m_i, d_i)$ slots.
3. All other pools are assigned rw_i slots.
4. The pools’ shares sum up to the total number of slots t .

The Fair Scheduler uses binary search to compute the correct r . We define a function $f(r)$ as the number of slots that would be used for a given r if conditions 1-3 above were met, and then find a value of r that makes $f(r) = t$. More precisely, $f(r)$ is defined as:

$$f(r) = \sum_i \min(d_i, \max(rw_i, m_i)).$$

Note that $f(r)$ is increasing in r because every term of the sum is increasing, so the equation $f(r) = t$ can be solved by binary search. We choose 0 as a lower bound of our binary search because with $r = 0$, only min shares are assigned. (An earlier check in `computeFairShares` checks whether the min shares add up to more than the total number of slots, and if so, computes fair shares by scaling down the min shares proportionally and returns.) To compute an upper bound for the binary search, we try $r = 1, 2, 4, 8, \dots$ until we find a value large enough that either more than t slots are used or all pools’ demands are met (in case the demands added up to less than t).

The steps of the algorithm are explained in detail in `SchedulingAlgorithms.java`.

This algorithm runs in time $O(NP)$, where N is the number of jobs/pools and P is the desired number of bits of precision in the computed values (number of iterations of binary search), which we’ve set to 25. It thus scales linearly in the number of jobs and pools.

4.7 Running Job Limits

Running job limits are implemented by marking jobs as not runnable if there are too many jobs submitted by the same user or pool. This is done in `FairScheduler.updateRunnability`. A job that is not runnable declares its demand as 0 and always returns null from `assignTasks`.

4.8 Delay Scheduling

In Hadoop, running map tasks on the nodes or racks that contain their input data is critical for performance, because it avoids shipping the data over the network. However, always assigning slots to the first job in order of pool shares and in-pool ordering (the “head-of-line job”) can sometimes lead to poor locality:

- If the head-of-line job is small, the chance of it having data on the node that a heartbeat was received from is small. Therefore, locality would be poor in a small-job workload if we always assigned slots to the head-of-line job.
- When fair sharing is used, there is a strong bias for a job to be reassigned into a slot that it just finished a task in, because when it finishes the task, the job falls below its fair share. This can mean that jobs have a difficult time running in slots that other jobs have taken and thus achieve poor locality.

To deal with both of these situations, the Fair Scheduler can sacrifice fairness temporarily to improve locality through an algorithm called delay scheduling. If the head-of-line job cannot launch a local task on the TaskTracker that sent a heartbeat, then it is skipped, and other running jobs are looked at in order of pool shares and in-pool scheduling rules to find a job with a local task. However, if the head-of-line job has been skipped for a sufficiently long time, it is allowed to launch rack-local tasks. Then, if it is skipped for a longer time, it is also allowed to launch off-rack tasks. These skip times are called locality delays. Delays of a few seconds are sufficient to drastically increase locality.

The Fair Scheduler allows locality delays to be set through `mapred-site.xml` or to be turned off by setting them to zero. However, by default, it computes the delay automatically based on the heartbeat interval of the cluster. The delay is set to 1.5x the heartbeat interval.

When a job that has been allowed to launch non-local tasks ends up launching a local task again, its “locality level” resets and it must wait again before launching non-local tasks. This is done so that a job that gets “unlucky” early in its lifetime does not continue to launch non-local tasks throughout its life.

Delay scheduling is implemented by keeping track of two variables on each job: the locality level of the last map it launched (0 for node-local, 1 for rack-local and 2 for off-rack) and the time it has spent being skipped for a task. These are kept in a `JobInfo` structure associated with each job in `FairScheduler.java`. Whenever a job is asked for tasks, it checks the locality level it is allowed to launch them at through `FairScheduler.getAllowedLocalityLevel`. If it does not launch a task, it is marked as “visited” on that heartbeat by appending itself to a `visited` job list that is passed around between calls to `assignTasks` on the same heartbeat. Jobs that are visited on a heartbeat but do not launch any tasks during it are considered as skipped for the time interval between this heartbeat and the next. Code at the beginning of `FairScheduler.assignTasks` increments the wait time of each skipped job by the time elapsed since the last heartbeat. Once a job has been skipped for more than the locality delay, `getAllowedLocalityLevel` starts returning higher locality so that

it is allowed to launch less-local tasks. Whenever the job launches a task, its wait time is reset, but we remember the locality level of the launched task so that the job is allowed to launch more tasks at this level without further waiting.

4.9 Locking Order

Fair Scheduler data structures can be touched by several threads. Most commonly, the JobTracker invokes `assignTasks`. This happens inside a block of code where the JobTracker has locked itself already. Therefore, to prevent deadlocks, we always ensure that *if both the FairScheduler and the JobTracker must be locked, the JobTracker is locked first*. Other threads that can lock the FairScheduler include the update thread and the web UI.

4.10 Unit Tests

The Fair Scheduler contains extensive unit tests using mock `TaskTrackerManager`, `JobInProgress`, `TaskInProgress`, and `Schedulable` objects. Scheduler tests are in `TestFairScheduler.java`. The `computeFairShares` algorithm is tested separately in `TestComputeFairShares.java`. All tests use accelerated time via a fake `Clock` class.

5 Code Guide

The following table lists some key source files in the Fair Scheduler:

File	Contents
<code>FairScheduler.java</code>	Scheduler entry point. Also contains update thread, and logic for preemption, delay scheduling, and running job limits.
<code>Schedulable.java</code>	Definition of the <code>Schedulable</code> class. Extended by <code>JobSchedulable</code> and <code>PoolSchedulable</code> .
<code>SchedulingAlgorithms.java</code>	Contains FIFO and fair sharing comparators, as well as the <code>computeFairShares</code> algorithm in Section 4.6.
<code>PoolManager.java</code>	Reads pool properties from the allocation file and maintains a collection of <code>Pool</code> objects. Pools are created on demand.
<code>Pool.java</code>	Represents a pool and stores its map and reduce <code>Schedulables</code> .
<code>FairSchedulerServlet.java</code>	Implements the scheduler's web UI.
<code>FairSchedulerEventLog.java</code>	An easy-to-parse event log for debugging. Must be enabled through <code>mapred.fairscheduler.eventlog.enabled</code> . If enabled, logs are placed in <code>\$HADOOP_LOG_DIR/fairscheduler</code> .
<code>TaskSelector.java</code>	A pluggable class responsible for picking tasks within a job. Currently, <code>DefaultTaskSelector</code> delegates to <code>JobInProgress</code> , but this would be a useful place to experiment with new algorithms for speculative execution and locality.
<code>LoadManager.java</code>	A pluggable class responsible for determining when to launch more tasks on a <code>TaskTracker</code> . Currently, <code>CapBasedLoadManager</code> uses slot counts, but this would be a useful place to experiment with scheduling based on machine load.
<code>WeightAdjuster.java</code>	A pluggable class responsible for setting job weights. An example, <code>NewJobWeightBooster</code> , is provided, which increases weight temporarily for new jobs.