Job Scheduling

Overview

Spark has several facilities for scheduling resources between computations. First, recall that, as described in the cluster mode overview, each Spark application (instance of SparkContext) runs an independent set of executor processes. The cluster managers that Spark runs on provide facilities for scheduling across applications. Second, *within* each Spark application, multiple "jobs" (Spark actions) may be running concurrently if they were submitted by different threads. This is common if your application is serving requests over the network; for example, the Shark server works this way. Spark includes a fair scheduler to schedule resources within each SparkContext.

Scheduling Across Applications

When running on a cluster, each Spark application gets an independent set of executor JVMs that only run tasks and store data for that application. If multiple users need to share your cluster, there are different options to manage allocation, depending on the cluster manager.

The simplest option, available on all cluster managers, is *static partitioning* of resources. With this approach, each application is given a maximum amount of resources it can use, and holds onto them for its whole duration. This is the approach used in Spark's standalone and YARN modes, as well as the coarse-grained Mesos mode. Resource allocation can be configured as follows, based on the cluster type:

- **Standalone mode:** By default, applications submitted to the standalone mode cluster will run in FIFO (first-in-first-out) order, and each application will try to use all available nodes. You can limit the number of nodes an application uses by setting the `spark.cores.max` configuration property in it, or change the default for applications that don't set this setting through `spark.deploy.defaultCores`. Finally, in addition to controlling cores, each application's `spark.executor.memory` setting controls its memory use.
- **Mesos:** To use static partitioning on Mesos, set the `spark.mesos.coarse` configuration property to `true`, and optionally set `spark.cores.max` to limit each application's resource share as in the standalone mode. You should also set `spark.executor.memory` to control the executor memory.
- **YARN:** The `--num-executors` option to the Spark YARN client controls how many executors it will allocate on the cluster, while `--executor-memory` and `--executor-cores` control the resources per executor.

A second option available on Mesos is *dynamic sharing* of CPU cores. In this mode, each Spark application still has a fixed and independent memory allocation (set by `spark.executor.memory`), but when the application is not running tasks on a machine, other applications may run tasks on those cores. This mode is useful when you expect large numbers of not overly active applications, such as shell sessions from separate users. However, it comes with a risk of less predictable latency, because it may take a while for an application to gain back cores on one node when it has work to do. To use this mode, simply use a `mesos://` URL without setting `spark.mesos.coarse` to true.

Note that none of the modes currently provide memory sharing across applications. If you would like to share data this way, we recommend running a single server application that can serve multiple requests by querying the same RDDs. For example, the Shark JDBC server works this way for SQL queries. In future releases, in-memory storage systems such as Tachyon will provide another approach to share RDDs.

Dynamic Resource Allocation

Spark 1.2 introduces the ability to dynamically scale the set of cluster resources allocated to your application up and down based on the workload. This means that your application may give resources back to the cluster if they are no longer used and request them again later when there is demand. This feature is particularly useful if multiple applications share resources in your Spark cluster. If a subset of the resources allocated to an application becomes idle, it can be returned to the cluster's pool of resources and acquired by other applications. In Spark, dynamic resource allocation is performed on the granularity of the executor and can be enabled through `spark.dynamicAllocation.enabled`.

This feature is currently disabled by default and available only on YARN. A future release will extend this to standalone mode and Mesos coarse-grained mode. Note that although Spark on Mesos already has a similar notion of dynamic resource sharing in fine-grained mode, enabling dynamic allocation allows your Mesos application to take advantage of coarse-grained low-latency scheduling while sharing cluster resources efficiently.

Configuration and Setup

All configurations used by this feature live under the `spark.dynamicAllocation.*` namespace. To enable this feature, your application must set `spark.dynamicAllocation.enabled` to `true` and provide lower and upper bounds for the number of executors through `spark.dynamicAllocation.minExecutors` and `spark.dynamicAllocation.maxExecutors`. Other relevant configurations are described on the configurations page and in the subsequent sections in detail.

Additionally, your application must use an external shuffle service. The purpose of the service is to preserve the shuffle files written by executors so the executors can be safely removed (more detail described below). To enable this service, set `spark.shuffle.service.enabled` to `true`. In YARN, this external shuffle service is implemented in `org.apache.spark.yarn.network.YarnShuffleService` that runs in each `NodeManager` in your cluster. To start this service, follow these steps:

1. Build Spark with the YARN profile. Skip this step if you are using a pre-packaged distribution.
2. Locate the `spark-<version>-yarn-shuffle.jar`. This should be under `$SPARK_HOME/network/yarn/target/scala-<version>` if you are building Spark yourself, and under `lib` if you are using a distribution.
3. Add this jar to the classpath of all `NodeManager`s in your cluster.
4. In the `yarn-site.xml` on each node, add `spark_shuffle` to `yarn.nodemanager.aux-services`, then set `yarn.nodemanager.aux-services.spark_shuffle.class` to `org.apache.spark.network.yarn.YarnShuffleService`. Additionally, set all relevant `spark.shuffle.service.*` configurations.
5. Restart all `NodeManager`s in your cluster.

Resource Allocation Policy

At a high level, Spark should relinquish executors when they are no longer used and acquire executors when they are needed. Since there is no definitive way to predict whether an executor that is about to be removed will run a task in the near future, or whether a new executor that is about to be added will actually be idle, we need a set of heuristics to determine when to remove and request executors.

Request Policy

A Spark application with dynamic allocation enabled requests additional executors when it has pending tasks waiting to be scheduled. This condition necessarily implies that the existing set of executors is insufficient to simultaneously saturate all tasks that have been submitted but not yet finished.

Spark requests executors in rounds. The actual request is triggered when there have been pending tasks for `spark.dynamicAllocation.schedulerBacklogTimeout` seconds, and then triggered again every `spark.dynamicAllocation.sustainedSchedulerBacklogTimeout` seconds thereafter if the queue of pending tasks persists. Additionally, the number of executors requested in each round increases exponentially from the previous round. For instance, an application will add 1 executor in the first round, and then 2, 4, 8 and so on executors in the subsequent rounds.

The motivation for an exponential increase policy is twofold. First, an application should request executors cautiously in the beginning in case it turns out that only a few additional executors is sufficient. This echoes the justification for TCP slow start. Second, the application should be able to ramp up its resource usage in a timely manner in case it turns out that many executors are actually needed.

Remove Policy

The policy for removing executors is much simpler. A Spark application removes an executor when it has been idle for more than `spark.dynamicAllocation.executorIdleTimeout` seconds. Note that, under most circumstances, this condition is mutually exclusive with the request condition, in that an executor should not be idle if there are still pending tasks to be scheduled.

Graceful Decommission of Executors

Before dynamic allocation, a Spark executor exits either on failure or when the associated application has also exited. In both scenarios, all state associated with the executor is no longer needed and can be safely discarded. With dynamic allocation, however, the application is still running when an executor is explicitly removed. If the application attempts to access state stored in or written by the executor, it will have to perform a recompute the state. Thus, Spark needs a mechanism to decommission an executor gracefully by preserving its state before removing it.

This requirement is especially important for shuffles. During a shuffle, the Spark executor first writes its own map outputs locally to disk, and then acts as the server for those files when other executors attempt to fetch them. In the event of stragglers, which are tasks that run for much longer than their peers, dynamic allocation may remove an executor before the shuffle completes, in which case the shuffle files written by that executor must be recomputed unnecessarily.

The solution for preserving shuffle files is to use an external shuffle service, also introduced in Spark 1.2. This service refers to a long-running process that runs on each node of your cluster independently of your Spark applications and their executors. If the service is enabled, Spark executors will fetch shuffle files from the service instead of from each other. This means any shuffle state written by an executor may continue to be served beyond the executor's lifetime.

In addition to writing shuffle files, executors also cache data either on disk or in memory. When an executor is removed, however, all cached data will no longer be accessible. There is currently not yet a solution for this in Spark 1.2. In future releases, the cached data may be preserved through an off-heap storage similar in spirit to how shuffle files are preserved through the external shuffle service.

Scheduling Within an Application

Inside a given Spark application (SparkContext instance), multiple parallel jobs can run simultaneously if they were submitted from separate threads. By "job", in this section, we mean a Spark action (e.g. `save`, `collect`) and any tasks that need to run to evaluate that action. Spark's scheduler is fully thread-safe and supports this use case to enable applications that serve multiple requests (e.g. queries for multiple users).

By default, Spark's scheduler runs jobs in FIFO fashion. Each job is divided into "stages" (e.g. map and reduce phases), and the first job gets priority on all available resources while its stages have tasks to launch, then the second job gets priority, etc. If the jobs at the head of the queue don't need to use the whole cluster, later jobs can start to run right away, but if the jobs at the head of the queue are large, then later jobs may be delayed significantly.

Starting in Spark 0.8, it is also possible to configure fair sharing between jobs. Under fair sharing, Spark assigns tasks between jobs in a "round robin" fashion, so that all jobs get a roughly equal share of cluster resources. This means that short jobs submitted while a long job is running can start receiving resources right away and still get good response times, without waiting for the long job to finish. This mode is best for multi-user settings.

To enable the fair scheduler, simply set the `spark.scheduler.mode` property to `FAIR` when configuring a SparkContext:

```
val conf = new SparkConf().setMaster(...).setAppName(...)
conf.set("spark.scheduler.mode", "FAIR")
val sc = new SparkContext(conf)
```

Fair Scheduler Pools

The fair scheduler also supports grouping jobs into *pools*, and setting different scheduling options (e.g. weight) for each pool. This can be useful to create a "high-priority" pool for more important jobs, for example, or to group the jobs of each user together and give *users* equal shares regardless of how many concurrent jobs they have instead of giving *jobs* equal shares. This approach is modeled after the Hadoop Fair Scheduler.

Without any intervention, newly submitted jobs go into a *default pool*, but jobs' pools can be set by adding the `spark.scheduler.pool` "local property" to the SparkContext in the thread that's submitting them. This is done as follows:

```
// Assuming sc is your SparkContext variable
sc.setLocalProperty("spark.scheduler.pool", "pool1")
```

After setting this local property, *all* jobs submitted within this thread (by calls in this thread to `RDD.save`, `count`, `collect`, etc) will use this pool name. The setting is per-thread to make it easy to have a thread run multiple jobs on behalf of the same user. If you'd like to clear the pool that a thread is associated with, simply call:

```
sc.setLocalProperty("spark.scheduler.pool", null)
```

Default Behavior of Pools

By default, each pool gets an equal share of the cluster (also equal in share to each job in the default pool), but inside each pool, jobs run in FIFO order. For example, if you create one pool per user, this means that each user will get an equal share of the cluster, and that each user's queries will run in order instead of later queries taking resources from that user's earlier ones.

Configuring Pool Properties

Specific pools' properties can also be modified through a configuration file. Each pool supports three properties:

- `schedulingMode`: This can be FIFO or FAIR, to control whether jobs within the pool queue up behind each other (the default) or share the pool's resources fairly.
- `weight`: This controls the pool's share of the cluster relative to other pools. By default, all pools have a weight of 1. If you give a specific pool a weight of 2, for example, it will get 2x more resources as other active pools. Setting a high weight such as 1000 also makes it possible to implement *priority* between pools—in essence, the weight-1000 pool will always get to launch tasks first whenever it has jobs active.
- `minShare`: Apart from an overall weight, each pool can be given a *minimum shares* (as a number of CPU cores) that the administrator would like it to have. The fair scheduler always attempts to meet all active pools' minimum shares before redistributing extra resources according to the weights. The `minShare` property can therefore be another way to ensure that a pool can always get up to a certain number of resources (e.g. 10 cores) quickly without giving it a high priority for the rest of the cluster. By default, each pool's `minShare` is 0.

The pool properties can be set by creating an XML file, similar to `conf/fairscheduler.xml.template`, and setting a `spark.scheduler.allocation.file` property in your SparkConf.

```
conf.set("spark.scheduler.allocation.file", "/path/to/file")
```

The format of the XML file is simply a `<pool>` element for each pool, with different elements within it for the various settings. For example:

```
<?xml version="1.0"?>
<allocations>
  <pool name="production">
    <schedulingMode>FAIR</schedulingMode>
    <weight>1</weight>
    <minShare>2</minShare>
  </pool>
  <pool name="test">
    <schedulingMode>FIFO</schedulingMode>
    <weight>2</weight>
    <minShare>3</minShare>
  </pool>
</allocations>
```

A full example is also available in `conf/fairscheduler.xml.template`. Note that any pools not configured in the XML file will simply get default values for all settings (scheduling mode FIFO, weight 1, and minShare 0).