Spark Standalone Mode

- Security
- Installing Spark Standalone to a Cluster
- Starting a Cluster Manually
- Cluster Launch Scripts
- Resource Allocation and Configuration Overview
- Connecting an Application to the Cluster
- Client Properties
- Launching Spark Applications
- Resource Scheduling
- Executors Scheduling
- Monitoring and Logging
- Running Alongside Hadoop
- Configuring Ports for Network Security
- High Availability
  - Standby Masters with ZooKeeper
  - Single-Node Recovery with Local File System

In addition to running on the Mesos or YARN cluster managers, Spark also provides a simple standalone deploy mode. You can launch a standalone cluster either manually, by starting a master and workers by hand, or use our provided launch scripts. It is also possible to run these daemons on a single machine for testing.

Security

Security features like authentication are not enabled by default. When deploying a cluster that is open to the internet or an untrusted network, it's important to secure access to the cluster to prevent unauthorized applications from running on the cluster. Please see Spark Security and the specific security sections in this doc before running Spark.

Installing Spark Standalone to a Cluster

To install Spark Standalone mode, you simply place a compiled version of Spark on each node on the cluster. You can obtain pre-built versions of Spark with each release or build it yourself.

Starting a Cluster Manually

You can start a standalone master server by executing:

```
./sbin/start-master.sh
```

Once started, the master will print out a `spark://HOST:PORT` URL for itself, which you can use to connect workers to it, or pass as the "master" argument to `SparkContext`. You can also find this URL on the master's web UI, which is http://localhost:8080 by default.

Similarly, you can start one or more workers and connect them to the master via:

```
./sbin/start-worker.sh <master-spark-URL>
```

Once you have started a worker, look at the master's web UI (http://localhost:8080 by default). You should see the new node listed there, along with its number of CPUs and memory (minus one gigabyte left for the OS).

Finally, the following configuration options can be passed to the master and worker:

Argument	Meaning
`-h HOST`, `--host HOST`	Hostname to listen on
`-i HOST`, `--ip HOST`	Hostname to listen on (deprecated, use -h or --host)
`-p PORT`, `--port PORT`	Port for service to listen on (default: 7077 for master, random for worker)
`--webui-port PORT`	Port for web UI (default: 8080 for master, 8081 for worker)
`-c CORES`, `--cores CORES`	Total CPU cores to allow Spark applications to use on the machine (default: all available); only on worker

---

`-m MEM`, `--memory MEM`	Total amount of memory to allow Spark applications to use on the machine, in a format like 1000M or 2G (default: your machine's total RAM minus 1 GiB); only on worker
`-d DIR`, `--work-dir DIR`	Directory to use for scratch space and job output logs (default: SPARK_HOME/work); only on worker
`--properties-file FILE`	Path to a custom Spark properties file to load (default: conf/spark-defaults.conf)

Cluster Launch Scripts

To launch a Spark standalone cluster with the launch scripts, you should create a file called conf/workers in your Spark directory, which must contain the hostnames of all the machines where you intend to start Spark workers, one per line. If conf/workers does not exist, the launch scripts defaults to a single machine (localhost), which is useful for testing. Note, the master machine accesses each of the worker machines via ssh. By default, ssh is run in parallel and requires password-less (using a private key) access to be setup. If you do not have a password-less setup, you can set the environment variable SPARK_SSH_FOREGROUND and serially provide a password for each worker.

Once you've set up this file, you can launch or stop your cluster with the following shell scripts, based on Hadoop's deploy scripts, and available in `SPARK_HOME/sbin`:

- `sbin/start-master.sh` - Starts a master instance on the machine the script is executed on.
- `sbin/start-workers.sh` - Starts a worker instance on each machine specified in the `conf/workers` file.
- `sbin/start-worker.sh` - Starts a worker instance on the machine the script is executed on.
- `sbin/start-all.sh` - Starts both a master and a number of workers as described above.
- `sbin/stop-master.sh` - Stops the master that was started via the `sbin/start-master.sh` script.
- `sbin/stop-worker.sh` - Stops all worker instances on the machine the script is executed on.
- `sbin/stop-workers.sh` - Stops all worker instances on the machines specified in the `conf/workers` file.
- `sbin/stop-all.sh` - Stops both the master and the workers as described above.

Note that these scripts must be executed on the machine you want to run the Spark master on, not your local machine.

You can optionally configure the cluster further by setting environment variables in `conf/spark-env.sh`. Create this file by starting with the `conf/spark-env.sh.template`, and *copy it to all your worker machines* for the settings to take effect. The following settings are available:

Environment Variable	**Meaning**
`SPARK_MASTER_HOST`	Bind the master to a specific hostname or IP address, for example a public one.
`SPARK_MASTER_PORT`	Start the master on a different port (default: 7077).
`SPARK_MASTER_WEBUI_PORT`	Port for the master web UI (default: 8080).
`SPARK_MASTER_OPTS`	Configuration properties that apply only to the master in the form "-Dx=y" (default: none). See below for a list of possible options.
`SPARK_LOCAL_DIRS`	Directory to use for "scratch" space in Spark, including map output files and RDDs that get stored on disk. This should be on a fast, local disk in your system. It can also be a comma-separated list of multiple directories on different disks.
`SPARK_WORKER_CORES`	Total number of cores to allow Spark applications to use on the machine (default: all available cores).
`SPARK_WORKER_MEMORY`	Total amount of memory to allow Spark applications to use on the machine, e.g. `1000m`, `2g` (default: total memory minus 1 GiB); note that each application's *individual* memory is configured using its `spark.executor.memory` property.
`SPARK_WORKER_PORT`	Start the Spark worker on a specific port (default: random).
`SPARK_WORKER_WEBUI_PORT`	Port for the worker web UI (default: 8081).
`SPARK_WORKER_DIR`	Directory to run applications in, which will include both logs and scratch space (default: SPARK_HOME/work).
`SPARK_WORKER_OPTS`	Configuration properties that apply only to the worker in the form "-Dx=y" (default: none). See below for a list of possible options.
`SPARK_DAEMON_MEMORY`	Memory to allocate to the Spark master and worker daemons themselves (default: 1g).

SPARK_DAEMON_JAVA_OPTS	JVM options for the Spark master and worker daemons themselves in the form "-Dx=y" (default: none).
SPARK_DAEMON_CLASSPATH	Classpath for the Spark master and worker daemons themselves (default: none).
SPARK_PUBLIC_DNS	The public DNS name of the Spark master and workers (default: none).

Note: The launch scripts do not currently support Windows. To run a Spark cluster on Windows, start the master and workers by hand.

SPARK_MASTER_OPTS supports the following system properties:

Property Name	Default	Meaning	Since Version
spark.deploy.retainedApplications	200	The maximum number of completed applications to display. Older applications will be dropped from the UI to maintain this limit.	0.8.0
spark.deploy.retainedDrivers	200	The maximum number of completed drivers to display. Older drivers will be dropped from the UI to maintain this limit.	1.1.0
spark.deploy.spreadOut	true	Whether the standalone cluster manager should spread applications out across nodes or try to consolidate them onto as few nodes as possible. Spreading out is usually better for data locality in HDFS, but consolidating is more efficient for compute-intensive workloads.	0.6.1
spark.deploy.defaultCores	(infinite)	Default number of cores to give to applications in Spark's standalone mode if they don't set spark.cores.max. If not set, applications always get all available cores unless they configure spark.cores.max themselves. Set this lower on a shared cluster to prevent users from grabbing the whole cluster by default.	0.9.0
spark.deploy.maxExecutorRetries	10	Limit on the maximum number of back-to-back executor failures that can occur before the standalone cluster manager removes a faulty application. An application will never be removed if it has any running executors. If an application experiences more than spark.deploy.maxExecutorRetries failures in a row, no executors successfully start running in between those failures, and the application has no running executors then the standalone cluster manager will remove the application and mark it as failed. To disable this automatic removal, set spark.deploy.maxExecutorRetries to -1.	1.6.3
spark.worker.timeout	60	Number of seconds after which the standalone deploy master considers a worker lost if it receives no heartbeats.	0.6.2
spark.worker.resource.{resourceName}.amount	(none)	Amount of a particular resource to use on the worker.	3.0.0
spark.worker.resource.{resourceName}.discoveryScript	(none)	Path to resource discovery script, which is used to find a particular resource while worker starting up. And the output of the script should be formatted like the ResourceInformation class.	3.0.0
spark.worker.resourcesFile	(none)	Path to resources file which is used to find various resources while worker starting up. The content of resources file should be formatted like [{"id":{"componentName": "spark.worker","resourceName":"gpu"},"addresses": ["0","1","2"]}]. If a particular resource is not found in the resources file, the discovery script would be used to find that resource. If the discovery script also does not find the resources, the worker will fail to start up.	3.0.0

SPARK_WORKER_OPTS supports the following system properties:

Property Name	Default	Meaning	Since Version

`spark.worker.cleanup.enabled`	false	Enable periodic cleanup of worker / application directories. Note that this only affects standalone mode, as YARN works differently. Only the directories of stopped applications are cleaned up. This should be enabled if spark.shuffle.service.db.enabled is "true"	1.0.0
`spark.worker.cleanup.interval`	1800 (30 minutes)	Controls the interval, in seconds, at which the worker cleans up old application work dirs on the local machine.	1.0.0
`spark.worker.cleanup.appDataTtl`	604800 (7 days, 7 * 24 * 3600)	The number of seconds to retain application work directories on each worker. This is a Time To Live and should depend on the amount of available disk space you have. Application logs and jars are downloaded to each application work dir. Over time, the work dirs can quickly fill up disk space, especially if you run jobs very frequently.	1.0.0
`spark.shuffle.service.db.enabled`	true	Store External Shuffle service state on local disk so that when the external shuffle service is restarted, it will automatically reload info on current executors. This only affects standalone mode (yarn always has this behavior enabled). You should also enable `spark.worker.cleanup.enabled`, to ensure that the state eventually gets cleaned up. This config may be removed in the future.	3.0.0
`spark.storage.cleanupFilesAfterExecutorExit`	true	Enable cleanup non-shuffle files(such as temp. shuffle blocks, cached RDD/broadcast blocks, spill files, etc) of worker directories following executor exits. Note that this doesn't overlap with `spark.worker.cleanup.enabled`, as this enables cleanup of non-shuffle files in local directories of a dead executor, while `spark.worker.cleanup.enabled` enables cleanup of all files/subdirectories of a stopped and timeout application. This only affects Standalone mode, support of other cluster managers can be added in the future.	2.4.0
`spark.worker.ui.compressedLogFileLengthCacheSize`	100	For compressed log files, the uncompressed file can only be computed by uncompressing the files. Spark caches the uncompressed file size of compressed log files. This property controls the cache size.	2.0.2

Resource Allocation and Configuration Overview

Please make sure to have read the Custom Resource Scheduling and Configuration Overview section on the configuration page. This section only talks about the Spark Standalone specific aspects of resource scheduling.

Spark Standalone has 2 parts, the first is configuring the resources for the Worker, the second is the resource allocation for a specific application.

The user must configure the Workers to have a set of resources available so that it can assign them out to Executors. The `spark.worker.resource.{resourceName}.amount` is used to control the amount of each resource the worker has allocated. The user must also specify either `spark.worker.resourcesFile` or `spark.worker.resource.{resourceName}.discoveryScript` to specify how the Worker discovers the resources its assigned. See the descriptions above for each of those to see which method works best for your setup.

The second part is running an application on Spark Standalone. The only special case from the standard Spark resource configs is when you are running the Driver in client mode. For a Driver in client mode, the user can specify the resources it uses via `spark.driver.resourcesFile` or `spark.driver.resource.{resourceName}.discoveryScript`. If the Driver is running on the same host as other Drivers, please make sure the resources file or discovery script only returns resources that do not conflict with other Drivers running on the same node.

Note, the user does not need to specify a discovery script when submitting an application as the Worker will start each Executor with the resources it allocates to it.

Connecting an Application to the Cluster

To run an application on the Spark cluster, simply pass the `spark://IP:PORT` URL of the master as to the `SparkContext` constructor.

To run an interactive Spark shell against the cluster, run the following command:

```
./bin/spark-shell --master spark://IP:PORT
```

You can also pass an option `--total-executor-cores <numCores>` to control the number of cores that spark-shell uses on the cluster.

Client Properties

Spark applications supports the following configuration properties specific to standalone mode:

Property Name	Default Value	Meaning	Since Version
`spark.standalone.submit.waitAppCompletion`	`false`	In standalone cluster mode, controls whether the client waits to exit until the application completes. If set to `true`, the client process will stay alive polling the driver's status. Otherwise, the client process will exit after submission.	3.1.0

Launching Spark Applications

The `spark-submit` script provides the most straightforward way to submit a compiled Spark application to the cluster. For standalone clusters, Spark currently supports two deploy modes. In `client` mode, the driver is launched in the same process as the client that submits the application. In `cluster` mode, however, the driver is launched from one of the Worker processes inside the cluster, and the client process exits as soon as it fulfills its responsibility of submitting the application without waiting for the application to finish.

If your application is launched through Spark submit, then the application jar is automatically distributed to all worker nodes. For any additional jars that your application depends on, you should specify them through the `--jars` flag using comma as a delimiter (e.g. `--jars jar1,jar2`). To control the application's configuration or execution environment, see Spark Configuration.

Additionally, standalone `cluster` mode supports restarting your application automatically if it exited with non-zero exit code. To use this feature, you may pass in the `--supervise` flag to `spark-submit` when launching your application. Then, if you wish to kill an application that is failing repeatedly, you may do so through:

```
./bin/spark-class org.apache.spark.deploy.Client kill <master url> <driver ID>
```

You can find the driver ID through the standalone Master web UI at `http://<master url>:8080`.

Resource Scheduling

The standalone cluster mode currently only supports a simple FIFO scheduler across applications. However, to allow multiple concurrent users, you can control the maximum number of resources each application will use. By default, it will acquire *all* cores in the cluster, which only makes sense if you just run one application at a time. You can cap the number of cores by setting `spark.cores.max` in your SparkConf. For example:

```
val conf = new SparkConf()
  .setMaster(...)
  .setAppName(...)
  .set("spark.cores.max", "10")
val sc = new SparkContext(conf)
```

In addition, you can configure `spark.deploy.defaultCores` on the cluster master process to change the default for applications that don't set `spark.cores.max` to something less than infinite. Do this by adding the following to `conf/spark-env.sh`:

```
export SPARK_MASTER_OPTS="-Dspark.deploy.defaultCores=<value>"
```

This is useful on shared clusters where users might not have configured a maximum number of cores individually.

Executors Scheduling

The number of cores assigned to each executor is configurable. When `spark.executor.cores` is explicitly set, multiple executors from the same application may be launched on the same worker if the worker has enough cores and memory. Otherwise, each executor grabs all the cores available on the worker by default, in which case only one executor per application may be launched on each worker during one single schedule iteration.

Monitoring and Logging

Spark's standalone mode offers a web-based user interface to monitor the cluster. The master and each worker has its own web UI that shows cluster and job statistics. By default, you can access the web UI for the master at port 8080. The port can be changed either in the configuration file or via command-line options.

In addition, detailed log output for each job is also written to the work directory of each worker node (`SPARK_HOME/work` by default). You will see two files for each job, `stdout` and `stderr`, with all output it wrote to its console.

Running Alongside Hadoop

You can run Spark alongside your existing Hadoop cluster by just launching it as a separate service on the same machines. To access Hadoop data from Spark, just use an hdfs:// URL (typically `hdfs://<namenode>:9000/path`, but you can find the right URL on your Hadoop Namenode's web UI). Alternatively, you can set up a separate cluster for Spark, and still have it access HDFS over the network; this will be slower than disk-local access, but may not be a concern if you are still running in the same local area network (e.g. you place a few Spark machines on each rack that you have Hadoop on).

Configuring Ports for Network Security

Generally speaking, a Spark cluster and its services are not deployed on the public internet. They are generally private services, and should only be accessible within the network of the organization that deploys Spark. Access to the hosts and ports used by Spark services should be limited to origin hosts that need to access the services.

This is particularly important for clusters using the standalone resource manager, as they do not support fine-grained access control in a way that other resource managers do.

For a complete list of ports to configure, see the security page.

High Availability

By default, standalone scheduling clusters are resilient to Worker failures (insofar as Spark itself is resilient to losing work by moving it to other workers). However, the scheduler uses a Master to make scheduling decisions, and this (by default) creates a single point of failure: if the Master crashes, no new applications can be created. In order to circumvent this, we have two high availability schemes, detailed below.

Standby Masters with ZooKeeper

Overview

Utilizing ZooKeeper to provide leader election and some state storage, you can launch multiple Masters in your cluster connected to the same ZooKeeper instance. One will be elected "leader" and the others will remain in standby mode. If the current leader dies, another Master will be elected, recover the old Master's state, and then resume scheduling. The entire recovery process (from the time the first leader goes down) should take between 1 and 2 minutes. Note that this delay only affects scheduling *new* applications – applications that were already running during Master failover are unaffected.

Learn more about getting started with ZooKeeper here.

Configuration

In order to enable this recovery mode, you can set SPARK_DAEMON_JAVA_OPTS in spark-env by configuring `spark.deploy.recoveryMode` and related spark.deploy.zookeeper.* configurations. For more information about these configurations please refer to the configuration doc

Possible gotcha: If you have multiple Masters in your cluster but fail to correctly configure the Masters to use ZooKeeper, the Masters will fail to discover each other and think they're all leaders. This will not lead to a healthy cluster state (as all Masters will schedule independently).

Details

After you have a ZooKeeper cluster set up, enabling high availability is straightforward. Simply start multiple Master processes on different nodes with the same ZooKeeper configuration (ZooKeeper URL and directory). Masters can be added and removed at any time.

In order to schedule new applications or add Workers to the cluster, they need to know the IP address of the current leader. This can be accomplished by simply passing in a list of Masters where you used to pass in a single one. For example, you might start your SparkContext pointing to `spark://host1:port1,host2:port2`. This would cause your SparkContext to try registering with both Masters – if `host1` goes down, this configuration would still be correct as we'd find the new leader, `host2`.

There's an important distinction to be made between "registering with a Master" and normal operation. When starting up, an application or Worker needs to be able to find and register with the current lead Master. Once it successfully registers, though, it is "in the system" (i.e., stored in ZooKeeper). If failover occurs, the new leader will contact all previously registered applications and Workers to inform them of the

change in leadership, so they need not even have known of the existence of the new Master at startup.

Due to this property, new Masters can be created at any time, and the only thing you need to worry about is that *new* applications and Workers can find it to register with in case it becomes the leader. Once registered, you're taken care of.

Single-Node Recovery with Local File System

Overview

ZooKeeper is the best way to go for production-level high availability, but if you just want to be able to restart the Master if it goes down, FILESYSTEM mode can take care of it. When applications and Workers register, they have enough state written to the provided directory so that they can be recovered upon a restart of the Master process.

Configuration

In order to enable this recovery mode, you can set SPARK_DAEMON_JAVA_OPTS in spark-env using this configuration:

System property	Meaning	Since Version
`spark.deploy.recoveryMode`	Set to FILESYSTEM to enable single-node recovery mode (default: NONE).	0.8.1
`spark.deploy.recoveryDirectory`	The directory in which Spark will store recovery state, accessible from the Master's perspective.	0.8.1

Details

- This solution can be used in tandem with a process monitor/manager like monit, or just to enable manual recovery via restart.
- While filesystem recovery seems straightforwardly better than not doing any recovery at all, this mode may be suboptimal for certain development or experimental purposes. In particular, killing a master via stop-master.sh does not clean up its recovery state, so whenever you start a new Master, it will enter recovery mode. This could increase the startup time by up to 1 minute if it needs to wait for all previously-registered Workers/clients to timeout.
- While it's not officially supported, you could mount an NFS directory as the recovery directory. If the original Master node dies completely, you could then start a Master on a different node, which would correctly recover all previously registered Workers/applications (equivalent to ZooKeeper recovery). Future applications will have to be able to find the new Master, however, in order to register.