Spark Configuration

Spark provides three locations to configure the system:

- Spark properties control most application parameters and can be set by using a SparkConf object, or through Java system properties.
- Environment variables can be used to set per-machine settings, such as the IP address, through the `conf/spark-env.sh` script on each node.
- Logging can be configured through `log4j.properties`.

Spark Properties

Spark properties control most application settings and are configured separately for each application. These properties can be set directly on a SparkConf passed to your `SparkContext`. `SparkConf` allows you to configure some of the common properties (e.g. master URL and application name), as well as arbitrary key-value pairs through the `set()` method. For example, we could initialize an application with two threads as follows:

Note that we run with local[2], meaning two threads - which represents "minimal" parallelism, which can help detect bugs that only exist when we run in a distributed context.

```
val conf = new SparkConf()
             .setMaster("local[2]")
             .setAppName("CountingSheep")
             .set("spark.executor.memory", "1g")
val sc = new SparkContext(conf)
```

Note that we can have more than 1 thread in local mode, and in cases like spark streaming, we may actually require one to prevent any sort of starvation issues.

Dynamically Loading Spark Properties

In some cases, you may want to avoid hard-coding certain configurations in a `SparkConf`. For instance, if you'd like to run the same application with different masters or different amounts of memory. Spark allows you to simply create an empty conf:

```
val sc = new SparkContext(new SparkConf())
```

Then, you can supply configuration values at runtime:

```
./bin/spark-submit --name "My app" --master local[4] --conf spark.shuffle.spill=false
  --conf "spark.executor.extraJavaOptions=-XX:+PrintGCDetails -XX:+PrintGCTimeStamps" myApp.jar
```

The Spark shell and `spark-submit` tool support two ways to load configurations dynamically. The first are command line options, such as `--master`, as shown above. `spark-submit` can accept any Spark property using the `--conf` flag, but uses special flags for properties that play a part in launching the Spark application. Running `./bin/spark-submit --help` will show the entire list of these options.

`bin/spark-submit` will also read configuration options from `conf/spark-defaults.conf`, in which each line consists of a key and a value separated by whitespace. For example:

```
spark.master            spark://5.6.7.8:7077
spark.executor.memory   512m
spark.eventLog.enabled  true
spark.serializer        org.apache.spark.serializer.KryoSerializer
```

Any values specified as flags or in the properties file will be passed on to the application and merged with those specified through SparkConf. Properties set directly on the SparkConf take highest precedence, then flags passed to `spark-submit` or `spark-shell`, then options in the `spark-defaults.conf` file. A few configuration keys have been renamed since earlier versions of Spark; in such cases, the older key names are still accepted, but take lower precedence than any instance of the newer key.

Viewing Spark Properties

The application web UI at `http://<driver>:4040` lists Spark properties in the "Environment" tab. This is a useful place to check to make sure that your properties have been set correctly. Note that only values explicitly specified through `spark-defaults.conf`, `SparkConf`, or the command line will appear. For all other configuration properties, you can assume the default value is used.

Available Properties

Most of the properties that control internal settings have reasonable default values. Some of the most common options to set are:

Application Properties

Property Name	Default	Meaning
`spark.app.name`	(none)	The name of your application. This will appear in the UI and in log data.
`spark.driver.cores`	1	Number of cores to use for the driver process, only in cluster mode.
`spark.driver.maxResultSize`	1g	Limit of total size of serialized results of all partitions for each Spark action (e.g. collect). Should be at least 1M, or 0 for unlimited. Jobs will be aborted if the total size is above this limit. Having a high limit may cause out-of-memory errors in driver (depends on spark.driver.memory and memory overhead of objects in JVM). Setting a proper limit can protect the driver from out-of-memory errors.
`spark.driver.memory`	512m	Amount of memory to use for the driver process, i.e. where SparkContext is initialized. (e.g. `512m`, `2g`). *Note:* In client mode, this config must not be set through the `SparkConf` directly in your application, because the driver JVM has already started at that point. Instead, please set this through the `--driver-memory` command line option or in your default properties file.
`spark.executor.memory`	512m	Amount of memory to use per executor process, in the same format as JVM memory strings (e.g. `512m`, `2g`).
`spark.extraListeners`	(none)	A comma-separated list of classes that implement `SparkListener`; when initializing SparkContext, instances of these classes will be created and registered with Spark's listener bus. If a class has a single-argument constructor that accepts a SparkConf, that constructor will be called; otherwise, a zero-argument constructor will be called. If no valid constructor can be found, the SparkContext creation will fail with an exception.
`spark.local.dir`	/tmp	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. NOTE: In Spark 1.0 and later this will be overriden by SPARK_LOCAL_DIRS (Standalone, Mesos) or LOCAL_DIRS (YARN) environment variables set by the cluster manager.
`spark.logConf`	false	Logs the effective SparkConf as INFO when a SparkContext is started.
`spark.master`	(none)	The cluster manager to connect to. See the list of allowed master URL's.

Apart from these, the following properties are also available, and may be useful in some situations:

Runtime Environment

Property Name	Default	Meaning
`spark.driver.extraClassPath`	(none)	Extra classpath entries to append to the classpath of the driver. *Note:* In client mode, this config must not be set through the `SparkConf` directly in your application, because the driver JVM has already started at that point. Instead, please set this through the `--driver-class-path` command line option or in your default properties file.
`spark.driver.extraJavaOptions`	(none)	A string of extra JVM options to pass to the driver. For instance, GC settings or other logging. *Note:* In client mode, this config must not be set through the `SparkConf` directly in your application, because the driver JVM has already started at that point. Instead, please set this through the `--driver-java-options` command line option or in your default properties file.
`spark.driver.extraLibraryPath`	(none)	Set a special library path to use when launching the driver JVM. *Note:* In client mode, this config must not be set through the `SparkConf` directly in your application, because the driver JVM has already started at that point.

		Instead, please set this through the `--driver-library-path` command line option or in your default properties file.
`spark.driver.userClassPathFirst`	false	(Experimental) Whether to give user-added jars precedence over Spark's own jars when loading classes in the the driver. This feature can be used to mitigate conflicts between Spark's dependencies and user dependencies. It is currently an experimental feature. This is used in cluster mode only.
`spark.executor.extraClassPath`	(none)	Extra classpath entries to append to the classpath of executors. This exists primarily for backwards-compatibility with older versions of Spark. Users typically should not need to set this option.
`spark.executor.extraJavaOptions`	(none)	A string of extra JVM options to pass to executors. For instance, GC settings or other logging. Note that it is illegal to set Spark properties or heap size settings with this option. Spark properties should be set using a SparkConf object or the spark-defaults.conf file used with the spark-submit script. Heap size settings can be set with spark.executor.memory.
`spark.executor.extraLibraryPath`	(none)	Set a special library path to use when launching executor JVM's.
`spark.executor.logs.rolling.maxRetainedFiles`	(none)	Sets the number of latest rolling log files that are going to be retained by the system. Older log files will be deleted. Disabled by default.
`spark.executor.logs.rolling.size.maxBytes`	(none)	Set the max size of the file by which the executor logs will be rolled over. Rolling is disabled by default. Value is set in terms of bytes. See `spark.executor.logs.rolling.maxRetainedFiles` for automatic cleaning of old logs.
`spark.executor.logs.rolling.strategy`	(none)	Set the strategy of rolling of executor logs. By default it is disabled. It can be set to "time" (time-based rolling) or "size" (size-based rolling). For "time", use `spark.executor.logs.rolling.time.interval` to set the rolling interval. For "size", use `spark.executor.logs.rolling.size.maxBytes` to set the maximum file size for rolling.
`spark.executor.logs.rolling.time.interval`	daily	Set the time interval by which the executor logs will be rolled over. Rolling is disabled by default. Valid values are \`daily\`, \`hourly\`, \`minutely\` or any interval in seconds. See `spark.executor.logs.rolling.maxRetainedFiles` for automatic cleaning of old logs.
`spark.executor.userClassPathFirst`	false	(Experimental) Same functionality as `spark.driver.userClassPathFirst`, but applied to executor instances.
`spark.executorEnv.[EnvironmentVariableName]`	(none)	Add the environment variable specified by `EnvironmentVariableName` to the Executor process. The user can specify multiple of these to set multiple environment variables.
`spark.python.profile`	false	Enable profiling in Python worker, the profile result will show up by \`sc.show_profiles()\`, or it will be displayed before the driver exiting. It also can be dumped into disk by \`sc.dump_profiles(path)\`. If some of the profile results had been displayed maually, they will not be displayed automatically before driver exiting. By default the \`pyspark.profiler.BasicProfiler\` will be used, but this can be overridden by passing a profiler class in as a parameter to the \`SparkContext\` constructor.
`spark.python.profile.dump`	(none)	The directory which is used to dump the profile result before driver exiting. The results will be dumped as separated file for each RDD. They can be loaded by ptats.Stats(). If this is specified, the profile result will not be displayed automatically.
`spark.python.worker.memory`	512m	Amount of memory to use per python worker process during aggregation, in the same format as JVM memory strings (e.g. `512m`, `2g`). If the memory used during aggregation goes above this amount, it will spill the data into disks.
`spark.python.worker.reuse`	true	Reuse Python worker or not. If yes, it will use a fixed number of Python workers, does not need to fork() a Python process for every tasks. It will be very useful if there is large broadcast, then the broadcast will not be needed to transfered from JVM to Python worker for every task.

Shuffle Behavior

Property Name	Default	Meaning
`spark.reducer.maxMbInFlight`	48	Maximum size (in megabytes) of map outputs to fetch simultaneously from each reduce task. Since each output requires us to create a buffer to receive it, this represents a fixed memory overhead per reduce task, so keep it small unless you have a large amount of memory.

`spark.shuffle.blockTransferService`	netty	Implementation to use for transferring shuffle and cached blocks between executors. There are two implementations available: `netty` and `nio`. Netty-based block transfer is intended to be simpler but equally efficient and is the default option starting in 1.2.
`spark.shuffle.compress`	true	Whether to compress map output files. Generally a good idea. Compression will use `spark.io.compression.codec`.
`spark.shuffle.consolidateFiles`	false	If set to "true", consolidates intermediate files created during a shuffle. Creating fewer files can improve filesystem performance for shuffles with large numbers of reduce tasks. It is recommended to set this to "true" when using ext4 or xfs filesystems. On ext3, this option might degrade performance on machines with many (>8) cores due to filesystem limitations.
`spark.shuffle.file.buffer.kb`	32	Size of the in-memory buffer for each shuffle file output stream, in kilobytes. These buffers reduce the number of disk seeks and system calls made in creating intermediate shuffle files.
`spark.shuffle.io.maxRetries`	3	(Netty only) Fetches that fail due to IO-related exceptions are automatically retried if this is set to a non-zero value. This retry logic helps stabilize large shuffles in the face of long GC pauses or transient network connectivity issues.
`spark.shuffle.io.numConnectionsPerPeer`	1	(Netty only) Connections between hosts are reused in order to reduce connection buildup for large clusters. For clusters with many hard disks and few hosts, this may result in insufficient concurrency to saturate all disks, and so users may consider increasing this value.
`spark.shuffle.io.preferDirectBufs`	true	(Netty only) Off-heap buffers are used to reduce garbage collection during shuffle and cache block transfer. For environments where off-heap memory is tightly limited, users may wish to turn this off to force all allocations from Netty to be on-heap.
`spark.shuffle.io.retryWait`	5	(Netty only) Seconds to wait between retries of fetches. The maximum delay caused by retrying is simply `maxRetries * retryWait`, by default 15 seconds.
`spark.shuffle.manager`	sort	Implementation to use for shuffling data. There are two implementations available: `sort` and `hash`. Sort-based shuffle is more memory-efficient and is the default option starting in 1.2.
`spark.shuffle.memoryFraction`	0.2	Fraction of Java heap to use for aggregation and cogroups during shuffles, if `spark.shuffle.spill` is true. At any given time, the collective size of all in-memory maps used for shuffles is bounded by this limit, beyond which the contents will begin to spill to disk. If spills are often, consider increasing this value at the expense of `spark.storage.memoryFraction`.
`spark.shuffle.sort.bypassMergeThreshold`	200	(Advanced) In the sort-based shuffle manager, avoid merge-sorting data if there is no map-side aggregation and there are at most this many reduce partitions.
`spark.shuffle.spill`	true	If set to "true", limits the amount of memory used during reduces by spilling data out to disk. This spilling threshold is specified by `spark.shuffle.memoryFraction`.
`spark.shuffle.spill.compress`	true	Whether to compress data spilled during shuffles. Compression will use `spark.io.compression.codec`.

Spark UI

Property Name	Default	Meaning
`spark.eventLog.compress`	false	Whether to compress logged events, if `spark.eventLog.enabled` is true.
`spark.eventLog.dir`	file:///tmp/spark-events	Base directory in which Spark events are logged, if `spark.eventLog.enabled` is true. Within this base directory, Spark creates a sub-directory for each application, and logs the events specific to the application in this directory. Users may want to set this to a unified location like an HDFS directory so history files can be read by the history server.
`spark.eventLog.enabled`	false	Whether to log Spark events, useful for reconstructing the Web UI after the application has finished.
`spark.ui.killEnabled`	true	Allows stages and corresponding jobs to be killed from the web ui.
`spark.ui.port`	4040	Port for your application's dashboard, which shows memory and workload data.
`spark.ui.retainedJobs`	1000	How many jobs the Spark UI and status APIs remember before garbage collecting.
`spark.ui.retainedStages`	1000	How many stages the Spark UI and status APIs remember before garbage collecting.

Compression and Serialization

Property Name	Default	Meaning
`spark.broadcast.compress`	true	Whether to compress broadcast variables before sending them. Generally a good idea.
`spark.closure.serializer`	org.apache.spark.serializer. JavaSerializer	Serializer class to use for closures. Currently only the Java serializer is supported.
`spark.io.compression.codec`	snappy	The codec used to compress internal data such as RDD partitions, broadcast variables and shuffle outputs. By default, Spark provides three codecs: `lz4`, `lzf`, and `snappy`. You can also use fully qualified class names to specify the codec, e.g. `org.apache.spark.io.LZ4CompressionCodec`, `org.apache.spark.io.LZFCompressionCodec`, and `org.apache.spark.io.SnappyCompressionCodec`.
`spark.io.compression.lz4.block.size`	32768	Block size (in bytes) used in LZ4 compression, in the case when LZ4 compression codec is used. Lowering this block size will also lower shuffle memory usage when LZ4 is used.
`spark.io.compression.snappy.block.size`	32768	Block size (in bytes) used in Snappy compression, in the case when Snappy compression codec is used. Lowering this block size will also lower shuffle memory usage when Snappy is used.
`spark.kryo.classesToRegister`	(none)	If you use Kryo serialization, give a comma-separated list of custom class names to register with Kryo. See the tuning guide for more details.
`spark.kryo.referenceTracking`	true	Whether to track references to the same object when serializing data with Kryo, which is necessary if your object graphs have loops and useful for efficiency if they contain multiple copies of the same object. Can be disabled to improve performance if you know this is not the case.
`spark.kryo.registrationRequired`	false	Whether to require registration with Kryo. If set to 'true', Kryo will throw an exception if an unregistered class is serialized. If set to false (the default), Kryo will write unregistered class names along with each object. Writing class names can cause significant performance overhead, so enabling this option can enforce strictly that a user has not omitted classes from registration.
`spark.kryo.registrator`	(none)	If you use Kryo serialization, set this class to register your custom classes with Kryo. This property is useful if you need to register your classes in a custom way, e.g. to specify a custom field serializer. Otherwise `spark.kryo.classesToRegister` is simpler. It should be set to a class that extends `KryoRegistrator`. See the tuning guide for more details.
`spark.kryoserializer.buffer.max.mb`	64	Maximum allowable size of Kryo serialization buffer, in megabytes. This must be larger than any object you attempt to serialize. Increase this if you get a "buffer limit exceeded" exception inside Kryo.
`spark.kryoserializer.buffer.mb`	0.064	Initial size of Kryo's serialization buffer, in megabytes. Note that there will be one buffer *per core* on each worker. This buffer will grow up to `spark.kryoserializer.buffer.max.mb` if needed.
`spark.rdd.compress`	false	Whether to compress serialized RDD partitions (e.g. for `StorageLevel.MEMORY_ONLY_SER`). Can save substantial space at the cost of some extra CPU time.
`spark.serializer`	org.apache.spark.serializer. JavaSerializer	Class to use for serializing objects that will be sent over the network or need to be cached in serialized form. The default of Java serialization works with any Serializable Java object but is quite slow, so we recommend using `org.apache.spark.serializer.KryoSerializer` and configuring Kryo serialization when speed is necessary. Can be any subclass of `org.apache.spark.Serializer`.
`spark.serializer.objectStreamReset`	100	When serializing using org.apache.spark.serializer.JavaSerializer, the serializer caches objects to prevent writing redundant data, however that stops garbage collection of those objects. By calling 'reset' you flush that info from the serializer, and allow old objects to be collected. To turn off this periodic reset set it to -1. By default it will reset the serializer every 100 objects.

Execution Behavior

Property Name	Default	Meaning
`spark.broadcast.blockSize`	4096	Size of each piece of a block in kilobytes for `TorrentBroadcastFactory`. Too large a value decreases parallelism during broadcast (makes it slower); however, if it is too small, `BlockManager` might take a performance hit.
`spark.broadcast.factory`	org.apache.spark.broadcast. TorrentBroadcastFactory	Which broadcast implementation to use.
`spark.cleaner.ttl`	(infinite)	Duration (seconds) of how long Spark will remember any metadata (stages generated, tasks generated, etc.). Periodic cleanups will ensure that metadata older than this duration will be forgotten. This is useful for running Spark for many hours / days (for example, running 24/7 in case of Spark Streaming applications). Note that any RDD that persists in memory for more than this duration will be cleared as well.
`spark.default.parallelism`	For distributed shuffle operations like `reduceByKey` and `join`, the largest number of partitions in a parent RDD. For operations like `parallelize` with no parent RDDs, it depends on the cluster manager: • Local mode: number of cores on the local machine • Mesos fine grained mode: 8 • Others: total number of cores on all executor nodes or 2, whichever is larger	Default number of partitions in RDDs returned by transformations like `join`, `reduceByKey`, and `parallelize` when not set by user.
`spark.executor.heartbeatInterval`	10000	Interval (milliseconds) between each executor's heartbeats to the driver. Heartbeats let the driver know that the executor is still alive and update it with metrics for in-progress tasks.
`spark.files.fetchTimeout`	60	Communication timeout to use when fetching files added through SparkContext.addFile() from the driver, in seconds.
`spark.files.overwrite`	false	Whether to overwrite files added through SparkContext.addFile() when the target file exists and its contents do not match those of the source.
`spark.hadoop.cloneConf`	false	If set to true, clones a new Hadoop `Configuration` object for each task. This option should be enabled to work around `Configuration` thread-safety issues (see SPARK-2546 for more details). This is disabled by default in order to avoid unexpected performance regressions for jobs that are not affected by these issues.
`spark.hadoop.validateOutputSpecs`	true	If set to true, validates the output specification (e.g. checking if the output directory already exists) used in saveAsHadoopFile and other variants. This can be disabled to silence exceptions due to pre-existing output directories. We recommend that users do not disable this except if trying to achieve compatibility with previous versions of Spark. Simply use Hadoop's FileSystem API to delete output directories by hand. This setting is ignored for jobs generated through Spark Streaming's StreamingContext, since data may need to be rewritten to pre-existing output directories during checkpoint recovery.
`spark.storage.memoryFraction`	0.6	Fraction of Java heap to use for Spark's memory cache. This should not be larger than the "old" generation of objects in the JVM, which by default is given 0.6 of the heap, but you can increase it if you configure your own old generation size.
`spark.storage.memoryMapThreshold`	2097152	Size of a block, in bytes, above which Spark memory maps when reading a block from disk. This prevents Spark from memory mapping very small blocks. In general, memory mapping has high overhead for blocks close to or below the page size of the operating system.
`spark.storage.unrollFraction`	0.2	Fraction of `spark.storage.memoryFraction` to use for unrolling blocks in memory. This is dynamically allocated by dropping existing blocks when there is not enough free storage space to

		unroll the new block in its entirety.
`spark.tachyonStore.baseDir`	System.getProperty("java.io.tmpdir")	Directories of the Tachyon File System that store RDDs. The Tachyon file system's URL is set by `spark.tachyonStore.url`. It can also be a comma-separated list of multiple directories on Tachyon file system.
`spark.tachyonStore.url`	tachyon://localhost:19998	The URL of the underlying Tachyon file system in the TachyonStore.

Networking

Property Name	Default	Meaning
`spark.akka.failure-detector.threshold`	300.0	This is set to a larger value to disable failure detector that comes inbuilt akka. It can be enabled again, if you plan to use this feature (Not recommended). This maps to akka's \`akka.remote.transport-failure-detector.threshold\`. Tune this in combination of \`spark.akka.heartbeat.pauses\` and \`spark.akka.heartbeat.interval\` if you need to.
`spark.akka.frameSize`	10	Maximum message size to allow in "control plane" communication (for serialized tasks and task results), in MB. Increase this if your tasks need to send back large results to the driver (e.g. using `collect()` on a large dataset).
`spark.akka.heartbeat.interval`	1000	This is set to a larger value to disable the transport failure detector that comes built in to Akka. It can be enabled again, if you plan to use this feature (Not recommended). A larger interval value in seconds reduces network overhead and a smaller value (~ 1 s) might be more informative for Akka's failure detector. Tune this in combination of \`spark.akka.heartbeat.pauses\` if you need to. A likely positive use case for using failure detector would be: a sensistive failure detector can help evict rogue executors quickly. However this is usually not the case as GC pauses and network lags are expected in a real Spark cluster. Apart from that enabling this leads to a lot of exchanges of heart beats between nodes leading to flooding the network with those.
`spark.akka.heartbeat.pauses`	6000	This is set to a larger value to disable the transport failure detector that comes built in to Akka. It can be enabled again, if you plan to use this feature (Not recommended). Acceptable heart beat pause in seconds for Akka. This can be used to control sensitivity to GC pauses. Tune this along with \`spark.akka.heartbeat.interval\` if you need to.
`spark.akka.threads`	4	Number of actor threads to use for communication. Can be useful to increase on large clusters when the driver has a lot of CPU cores.
`spark.akka.timeout`	100	Communication timeout between Spark nodes, in seconds.
`spark.blockManager.port`	(random)	Port for all block managers to listen on. These exist on both the driver and the executors.
`spark.broadcast.port`	(random)	Port for the driver's HTTP broadcast server to listen on. This is not relevant for torrent broadcast.
`spark.driver.host`	(local hostname)	Hostname or IP address for the driver to listen on. This is used for communicating with the executors and the standalone Master.
`spark.driver.port`	(random)	Port for the driver to listen on. This is used for communicating with the executors and the standalone Master.
`spark.executor.port`	(random)	Port for the executor to listen on. This is used for communicating with the driver.
`spark.fileserver.port`	(random)	Port for the driver's HTTP file server to listen on.
`spark.network.timeout`	120	Default timeout for all network interactions, in seconds. This config will be used in place of `spark.core.connection.ack.wait.timeout`, `spark.akka.timeout`, `spark.storage.blockManagerSlaveTimeoutMs` or `spark.shuffle.io.connectionTimeout`, if they are not configured.
`spark.port.maxRetries`	16	Default maximum number of retries when binding to a port before giving up.
`spark.replClassServer.port`	(random)	Port for the driver's HTTP class server to listen on. This is only relevant for the Spark shell.

Scheduling

Property Name	Default	Meaning
`spark.cores.max`	(not set)	When running on a standalone deploy cluster or a Mesos cluster in "coarse-grained" sharing mode, the maximum amount of CPU cores to request for the application from across the cluster (not from each machine). If not set, the default will be `spark.deploy.defaultCores` on Spark's standalone cluster

		manager, or infinite (all available cores) on Mesos.
`spark.localExecution.enabled`	false	Enables Spark to run certain jobs, such as first() or take() on the driver, without sending tasks to the cluster. This can make certain jobs execute very quickly, but may require shipping a whole partition of data to the driver.
`spark.locality.wait`	3000	Number of milliseconds to wait to launch a data-local task before giving up and launching it on a less-local node. The same wait will be used to step through multiple locality levels (process-local, node-local, rack-local and then any). It is also possible to customize the waiting time for each level by setting `spark.locality.wait.node`, etc. You should increase this setting if your tasks are long and see poor locality, but the default usually works well.
`spark.locality.wait.node`	spark.locality.wait	Customize the locality wait for node locality. For example, you can set this to 0 to skip node locality and search immediately for rack locality (if your cluster has rack information).
`spark.locality.wait.process`	spark.locality.wait	Customize the locality wait for process locality. This affects tasks that attempt to access cached data in a particular executor process.
`spark.locality.wait.rack`	spark.locality.wait	Customize the locality wait for rack locality.
`spark.scheduler.maxRegisteredResourcesWaitingTime`	30000	Maximum amount of time to wait for resources to register before scheduling begins (in milliseconds).
`spark.scheduler.minRegisteredResourcesRatio`	0.8 for YARN mode; 0.0 otherwise	The minimum ratio of registered resources (registered resources / total expected resources) (resources are executors in yarn mode, CPU cores in standalone mode) to wait for before scheduling begins. Specified as a double between 0.0 and 1.0. Regardless of whether the minimum ratio of resources has been reached, the maximum amount of time it will wait before scheduling begins is controlled by config `spark.scheduler.maxRegisteredResourcesWaitingTime`.
`spark.scheduler.mode`	FIFO	The scheduling mode between jobs submitted to the same SparkContext. Can be set to `FAIR` to use fair sharing instead of queueing jobs one after another. Useful for multi-user services.
`spark.scheduler.revive.interval`	1000	The interval length for the scheduler to revive the worker resource offers to run tasks (in milliseconds).
`spark.speculation`	false	If set to "true", performs speculative execution of tasks. This means if one or more tasks are running slowly in a stage, they will be re-launched.
`spark.speculation.interval`	100	How often Spark will check for tasks to speculate, in milliseconds.
`spark.speculation.multiplier`	1.5	How many times slower a task is than the median to be considered for speculation.
`spark.speculation.quantile`	0.75	Percentage of tasks which must be complete before speculation is enabled for a particular stage.
`spark.task.cpus`	1	Number of cores to allocate for each task.
`spark.task.maxFailures`	4	Number of individual task failures before giving up on the job. Should be greater than or equal to 1. Number of allowed retries = this value - 1.

Dynamic Allocation

Property Name	Default	Meaning
`spark.dynamicAllocation.enabled`	false	Whether to use dynamic resource allocation, w scales the number of executors registered with application up and down based on the workloa Note that this is currently only available on YAF mode. For more detail, see the description her This requires `spark.shuffle.service.enabled` t set. The following configurations are also releva `spark.dynamicAllocation.minExecutors`, `spark.dynamicAllocation.maxExecutors`, and

		`spark.dynamicAllocation.initialExecutors`
`spark.dynamicAllocation.executorIdleTimeout`	600	If dynamic allocation is enabled and an executo been idle for more than this duration (in second the executor will be removed. For more detail, this description.
`spark.dynamicAllocation.initialExecutors`	`spark.dynamicAllocation.minExecutors`	Initial number of executors to run if dynamic allocation is enabled.
`spark.dynamicAllocation.maxExecutors`	Integer.MAX_VALUE	Upper bound for the number of executors if dy allocation is enabled.
`spark.dynamicAllocation.minExecutors`	0	Lower bound for the number of executors if dy allocation is enabled.
`spark.dynamicAllocation.schedulerBacklogTimeout`	5	If dynamic allocation is enabled and there have been pending tasks backlogged for more than duration (in seconds), new executors will be requested. For more detail, see this descriptior
`spark.dynamicAllocation.sustainedSchedulerBacklogTimeout`	`schedulerBacklogTimeout`	Same as `spark.dynamicAllocation.schedulerBacklogTi` but used only for subsequent executor request more detail, see this description.

Security

Property Name	Default	Meaning
`spark.acls.enable`	false	Whether Spark acls should are enabled. If enabled, this checks to see if the user has access permissions to view or modify the job. Note this requires the user to be known, so if the user comes across as null no checks are done. Filters can be used with the UI to authenticate and set the user.
`spark.admin.acls`	Empty	Comma separated list of users/administrators that have view and modify access to all Spark jobs. This can be used if you run on a shared cluster and have a set of administrators or devs who help debug when things work.
`spark.authenticate`	false	Whether Spark authenticates its internal connections. See `spark.authenticate.secret` if not running on YARN.
`spark.authenticate.secret`	None	Set the secret key used for Spark to authenticate between components. This needs to be set if not running on YARN and authentication is enabled.
`spark.core.connection.ack.wait.timeout`	60	Number of seconds for the connection to wait for ack to occur before timing out and giving up. To avoid unwilling timeout caused by long pause like GC, you can set larger value.
`spark.core.connection.auth.wait.timeout`	30	Number of seconds for the connection to wait for authentication to occur before timing out and giving up.
`spark.modify.acls`	Empty	Comma separated list of users that have modify access to the Spark job. By default only the user that started the Spark job has access to modify it (kill it for example).
`spark.ui.filters`	None	Comma separated list of filter class names to apply to the Spark web UI. The filter should be a standard javax servlet Filter. Parameters to each filter can also be specified by setting a java system property of: `spark.<class name of filter>.params='param1=value1,param2=value2'` For example: `-Dspark.ui.filters=com.test.filter1` `-Dspark.com.test.filter1.params='param1=foo,param2=testing'`
`spark.ui.view.acls`	Empty	Comma separated list of users that have view access to the Spark web ui. By default only the user that started the Spark job has view access.

Encryption

Property Name	Default	Meaning
`spark.ssl.enabled`	false	Whether to enable SSL connections on all supported protocols. All the SSL settings like `spark.ssl.xxx` where `xxx` is a particular configuration property, denote the global configuration for all the supported protocols. In order to override the global configuration for the particular protocol, the properties must be overwritten in the protocol-specific namespace. Use `spark.ssl.YYY.XXX` settings to overwrite the global configuration for particular protocol

		denoted by `YYY`. Currently `YYY` can be either `akka` for Akka based connections or `fs` for broadcast and file server.
`spark.ssl.enabledAlgorithms`	Empty	A comma separated list of ciphers. The specified ciphers must be supported by JVM. The reference list of protocols one can find on this page.
`spark.ssl.keyPassword`	None	A password to the private key in key-store.
`spark.ssl.keyStore`	None	A path to a key-store file. The path can be absolute or relative to the directory where the component is started in.
`spark.ssl.keyStorePassword`	None	A password to the key-store.
`spark.ssl.protocol`	None	A protocol name. The protocol must be supported by JVM. The reference list of protocols one can find on this page.
`spark.ssl.trustStore`	None	A path to a trust-store file. The path can be absolute or relative to the directory where the component is started in.
`spark.ssl.trustStorePassword`	None	A password to the trust-store.

Spark Streaming

Property Name	Default	Meaning
`spark.streaming.blockInterval`	200	Interval (milliseconds) at which data received by Spark Streaming receivers is chunked into blocks of data before storing them in Spark. Minimum recommended - 50 ms. See the performance tuning section in the Spark Streaming programing guide for more details.
`spark.streaming.receiver.maxRate`	not set	Maximum rate (number of records per second) at which each receiver will receive data. Effectively, each stream will consume at most this number of records per second. Setting this configuration to 0 or a negative number will put no limit on the rate. See the deployment guide in the Spark Streaming programing guide for mode details.
`spark.streaming.receiver.writeAheadLog.enable`	false	Enable write ahead logs for receivers. All the input data received through receivers will be saved to write ahead logs that will allow it to be recovered after driver failures. See the deployment guide in the Spark Streaming programing guide for more details.
`spark.streaming.unpersist`	true	Force RDDs generated and persisted by Spark Streaming to be automatically unpersisted from Spark's memory. The raw input data received by Spark Streaming is also automatically cleared. Setting this to false will allow the raw data and persisted RDDs to be accessible outside the streaming application as they will not be cleared automatically. But it comes at the cost of higher memory usage in Spark.
`spark.streaming.kafka.maxRatePerPartition`	not set	Maximum rate (number of records per second) at which data will be read from each Kafka partition when using the new Kafka direct stream API. See the Kafka Integration guide for more details.

Cluster Managers

Each cluster manager in Spark has additional configuration options. Configurations can be found on the pages for each mode:

- YARN
- Mesos
- Standalone Mode

Environment Variables

Certain Spark settings can be configured through environment variables, which are read from the `conf/spark-env.sh` script in the directory where Spark is installed (or `conf/spark-env.cmd` on Windows). In Standalone and Mesos modes, this file can give machine specific information such as hostnames. It is also sourced when running local Spark applications or submission scripts.

Note that `conf/spark-env.sh` does not exist by default when Spark is installed. However, you can copy `conf/spark-env.sh.template` to create it. Make sure you make the copy executable.

The following variables can be set in `spark-env.sh`:

Environment Variable	Meaning
`JAVA_HOME`	Location where Java is installed (if it's not on your default `PATH`).
`PYSPARK_PYTHON`	Python binary executable to use for PySpark.

`SPARK_LOCAL_IP`	IP address of the machine to bind to.
`SPARK_PUBLIC_DNS`	Hostname your Spark program will advertise to other machines.

In addition to the above, there are also options for setting up the Spark standalone cluster scripts, such as number of cores to use on each machine and maximum memory.

Since `spark-env.sh` is a shell script, some of these can be set programmatically – for example, you might compute `SPARK_LOCAL_IP` by looking up the IP of a specific network interface.

Configuring Logging

Spark uses log4j for logging. You can configure it by adding a `log4j.properties` file in the `conf` directory. One way to start is to copy the existing `log4j.properties.template` located there.

Overriding configuration directory

To specify a different configuration directory other than the default "SPARK_HOME/conf", you can set SPARK_CONF_DIR. Spark will use the the configuration files (spark-defaults.conf, spark-env.sh, log4j.properties, etc) from this directory.