Spark Configuration

- Spark Properties
  - Dynamically Loading Spark Properties
  - Viewing Spark Properties
  - Available Properties
    - Application Properties
    - Runtime Environment
    - Shuffle Behavior
    - Spark UI
    - Compression and Serialization
    - Memory Management
    - Execution Behavior
    - Networking
    - Scheduling
    - Dynamic Allocation
    - Security
    - TLS / SSL
    - Spark SQL
    - Spark Streaming
    - SparkR
    - Deploy
    - Cluster Managers
      - YARN
      - Mesos
      - Standalone Mode
- Environment Variables
- Configuring Logging
- Overriding configuration directory
- Inheriting Hadoop Cluster 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")
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 more than 1 thread to prevent any sort of starvation issues.

Properties that specify some time duration should be configured with a unit of time. The following format is accepted:

```
25ms (milliseconds)
5s (seconds)
10m or 10min (minutes)
3h (hours)
5d (days)
1y (years)
```

Properties that specify a byte size should be configured with a unit of size. The following format is accepted:

```
1b (bytes)
1k or 1kb (kibibytes = 1024 bytes)
1m or 1mb (mebibytes = 1024 kibibytes)
1g or 1gb (gibibytes = 1024 mebibytes)
```

```
1t or 1tb (tebibytes = 1024 gibibytes)
1p or 1pb (pebibytes = 1024 tebibytes)
```

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.eventLog.enabled=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   4g
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`	1g	Amount of memory to use for the driver process, i.e. where SparkContext is initialized. (e.g. `1g`, `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`	1g	Amount of memory to use per executor process (e.g. `2g`, `8g`).
`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 overridden 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.
`spark.submit.deployMode`	(none)	The deploy mode of Spark driver program, either "client" or "cluster", Which means to launch driver program locally ("client") or remotely ("cluster") on one of the nodes inside the cluster.
`spark.log.callerContext`	(none)	Application information that will be written into Yarn RM log/HDFS audit log when running on Yarn/HDFS. Its length depends on the Hadoop configuration `hadoop.caller.context.max.size`. It should be concise, and typically can have up to 50 characters.

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 prepend 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 that it is illegal to set maximum heap size (-Xmx) settings with this option. Maximum heap size settings can be set with `spark.driver.memory` in the cluster mode and through the `--driver-memory` command line option in the client mode. *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 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 prepend 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 maximum heap size (-Xmx) 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. Maximum 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.enableCompression`	false	Enable executor log compression. If it is enabled, the rolled executor logs will be compressed. Disabled by default.
`spark.executor.logs.rolling.maxSize`	(none)	Set the max size of the file in bytes by which the executor logs will be rolled over. Rolling is disabled by default. 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.maxSize` 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.redaction.regex`	(?i)secret|password	Regex to decide which Spark configuration properties and environment variables in driver and executor environments contain sensitive information. When this regex matches a property, its value is redacted from the environment UI and various logs like YARN and event logs.
`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 manually, 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 transferred from JVM to Python worker for every task.
`spark.files`		Comma-separated list of files to be placed in the working directory of each executor.
`spark.submit.pyFiles`		Comma-separated list of .zip, .egg, or .py files to place on the PYTHONPATH for Python apps.
`spark.jars`		Comma-separated list of local jars to include on the driver and executor classpaths.
`spark.jars.packages`		Comma-separated list of maven coordinates of jars to include on the driver and executor classpaths. Will search the local maven repo, then maven central and any additional remote repositories given by `spark.jars.ivy`. The format for the coordinates should be groupId:artifactId:version.
`spark.jars.excludes`		Comma-separated list of groupId:artifactId, to exclude while resolving the dependencies provided in `spark.jars.packages` to avoid dependency conflicts.

`spark.jars.ivy`	Comma-separated list of additional remote repositories to search for the coordinates given with `spark.jars.packages`.
`spark.pyspark.driver.python`	Python binary executable to use for PySpark in driver. (default is `spark.pyspark.python`)
`spark.pyspark.python`	Python binary executable to use for PySpark in both driver and executors.

Shuffle Behavior

Property Name	Default	Meaning
`spark.reducer.maxSizeInFlight`	48m	Maximum size 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.reducer.maxReqsInFlight`	Int.MaxValue	This configuration limits the number of remote requests to fetch blocks at any given point. When the number of hosts in the cluster increase, it might lead to very large number of in-bound connections to one or more nodes, causing the workers to fail under load. By allowing it to limit the number of fetch requests, this scenario can be mitigated.
`spark.shuffle.compress`	true	Whether to compress map output files. Generally a good idea. Compression will use `spark.io.compression.codec`.
`spark.shuffle.file.buffer`	32k	Size of the in-memory buffer for each shuffle file output stream. 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`	5s	(Netty only) How long to wait between retries of fetches. The maximum delay caused by retrying is 15 seconds by default, calculated as `maxRetries * retryWait`.
`spark.shuffle.service.enabled`	false	Enables the external shuffle service. This service preserves the shuffle files written by executors so the executors can be safely removed. This must be enabled if `spark.dynamicAllocation.enabled` is "true". The external shuffle service must be set up in order to enable it. See dynamic allocation configuration and setup documentation for more information.
`spark.shuffle.service.port`	7337	Port on which the external shuffle service will run.
`spark.shuffle.service.index.cache.entries`	1024	Max number of entries to keep in the index cache of the shuffle service.
`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.compress`	true	Whether to compress data spilled during shuffles. Compression will use `spark.io.compression.codec`.
`spark.io.encryption.enabled`	false	Enable IO encryption. Currently supported by all modes except Mesos. It's recommended that RPC encryption be enabled when using this feature.
`spark.io.encryption.keySizeBits`	128	IO encryption key size in bits. Supported values are 128, 192 and 256.
`spark.io.encryption.keygen.algorithm`	HmacSHA1	The algorithm to use when generating the IO encryption key. The supported algorithms are described in the KeyGenerator section of the Java Cryptography Architecture Standard Algorithm Name Documentation.

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.enabled`	true	Whether to run the web UI for the Spark application.
`spark.ui.killEnabled`	true	Allows jobs and stages 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.
`spark.ui.retainedTasks`	100000	How many tasks the Spark UI and status APIs remember before garbage collecting.
`spark.ui.reverseProxy`	false	Enable running Spark Master as reverse proxy for worker and application UIs. In this mode, Spark master will reverse proxy the worker and application UIs to enable access without requiring direct access to their hosts. Use it with caution, as worker and application UI will not be accessible directly, you will only be able to access them through spark master/proxy public URL. This setting affects all the workers and application UIs running in the cluster and must be set on all the workers, drivers and masters.
`spark.ui.reverseProxyUrl`		This is the URL where your proxy is running. This URL is for proxy which is running in front of Spark Master. This is useful when running proxy for authentication e.g. OAuth proxy. Make sure this is a complete URL including scheme (http/https) and port to reach your proxy.
`spark.ui.showConsoleProgress`	true	Show the progress bar in the console. The progress bar shows the progress of stages that run for longer than 500ms. If multiple stages run at the same time, multiple progress bars will be displayed on the same line.
`spark.worker.ui.retainedExecutors`	1000	How many finished executors the Spark UI and status APIs remember before garbage collecting.
`spark.worker.ui.retainedDrivers`	1000	How many finished drivers the Spark UI and status APIs remember before garbage collecting.
`spark.sql.ui.retainedExecutions`	1000	How many finished executions the Spark UI and status APIs remember before garbage collecting.
`spark.streaming.ui.retainedBatches`	1000	How many finished batches the Spark UI and status APIs remember before garbage collecting.
`spark.ui.retainedDeadExecutors`	100	How many dead executors 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.io.compression.codec`	lz4	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.blockSize`	32k	Block size 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.blockSize`	32k	Block size 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, give a comma-separated list of classes that 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 classes that extend `KryoRegistrator`. See the tuning guide for more details.
`spark.kryo.unsafe`	false	Whether to use unsafe based Kryo serializer. Can be substantially faster by using Unsafe Based IO.
`spark.kryoserializer.buffer.max`	64m	Maximum allowable size of Kryo serialization buffer. 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`	64k	Initial size of Kryo's serialization buffer. Note that there will be one buffer *per core* on each worker. This buffer will grow up to `spark.kryoserializer.buffer.max` if needed.
`spark.rdd.compress`	false	Whether to compress serialized RDD partitions (e.g. for `StorageLevel.MEMORY_ONLY_SER` in Java and Scala or `StorageLevel.MEMORY_ONLY` in Python). 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.

Memory Management

Property Name	Default	Meaning
`spark.memory.fraction`	0.6	Fraction of (heap space - 300MB) used for execution and storage. The lower this is, the more frequently spills and cached data eviction occur. The purpose of this config is to set aside memory for internal metadata, user data structures, and imprecise size estimation in the case of sparse,

		unusually large records. Leaving this at the default value is recommended. For more detail, including important information about correctly tuning JVM garbage collection when increasing this value, see this description.
`spark.memory.storageFraction`	0.5	Amount of storage memory immune to eviction, expressed as a fraction of the size of the region set aside by `spark.memory.fraction`. The higher this is, the less working memory may be available to execution and tasks may spill to disk more often. Leaving this at the default value is recommended. For more detail, see this description.
`spark.memory.offHeap.enabled`	false	If true, Spark will attempt to use off-heap memory for certain operations. If off-heap memory use is enabled, then `spark.memory.offHeap.size` must be positive.
`spark.memory.offHeap.size`	0	The absolute amount of memory in bytes which can be used for off-heap allocation. This setting has no impact on heap memory usage, so if your executors' total memory consumption must fit within some hard limit then be sure to shrink your JVM heap size accordingly. This must be set to a positive value when `spark.memory.offHeap.enabled=true`.
`spark.memory.useLegacyMode`	false	Whether to enable the legacy memory management mode used in Spark 1.5 and before. The legacy mode rigidly partitions the heap space into fixed-size regions, potentially leading to excessive spilling if the application was not tuned. The following deprecated memory fraction configurations are not read unless this is enabled: `spark.shuffle.memoryFraction` `spark.storage.memoryFraction` `spark.storage.unrollFraction`
`spark.shuffle.memoryFraction`	0.2	(deprecated) This is read only if `spark.memory.useLegacyMode` is enabled. Fraction of Java heap to use for aggregation and cogroups during shuffles. 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.storage.memoryFraction`	0.6	(deprecated) This is read only if `spark.memory.useLegacyMode` is enabled. 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.unrollFraction`	0.2	(deprecated) This is read only if `spark.memory.useLegacyMode` is enabled. 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.

Execution Behavior

Property Name	Default	Meaning
`spark.broadcast.blockSize`	4m	Size of each piece of a block 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.executor.cores`	1 in YARN mode, all the available cores on the worker in standalone and Mesos coarse-grained modes.	The number of cores to use on each executor. In standalone and Mesos coarse-grained modes, setting this parameter allows an application to run multiple executors on the same worker, provided that there are enough cores on that worker. Otherwise, only one executor per application will run on each worker.
`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	Default number of partitions in RDDs returned by transformations like `join`, `reduceByKey`, and `parallelize` when not set by user.

	larger	
`spark.executor.heartbeatInterval`	10s	Interval 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.executor.heartbeatInterval should be significantly less than spark.network.timeout
`spark.files.fetchTimeout`	60s	Communication timeout to use when fetching files added through SparkContext.addFile() from the driver.
`spark.files.useFetchCache`	true	If set to true (default), file fetching will use a local cache that is shared by executors that belong to the same application, which can improve task launching performance when running many executors on the same host. If set to false, these caching optimizations will be disabled and all executors will fetch their own copies of files. This optimization may be disabled in order to use Spark local directories that reside on NFS filesystems (see SPARK-6313 for more details).
`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.files.maxPartitionBytes`	134217728 (128 MB)	The maximum number of bytes to pack into a single partition when reading files.
`spark.files.openCostInBytes`	4194304 (4 MB)	The estimated cost to open a file, measured by the number of bytes could be scanned in the same time. This is used when putting multiple files into a partition. It is better to over estimate, then the partitions with small files will be faster than partitions with bigger files.
`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.memoryMapThreshold`	2m	Size of a block 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.

Networking

Property Name	Default	Meaning
`spark.rpc.message.maxSize`	128	Maximum message size (in MB) to allow in "control plane" communication; generally only applies to map output size information sent between executors and the driver. Increase this if you are running jobs with many thousands of map and reduce tasks and see messages about the RPC message size.
`spark.blockManager.port`	(random)	Port for all block managers to listen on. These exist on both the driver and the executors.
`spark.driver.blockManager.port`	(value of spark.blockManager.port)	Driver-specific port for the block manager to listen on, for cases where it cannot use the same configuration as executors.
`spark.driver.bindAddress`	(value of spark.driver.host)	Hostname or IP address where to bind listening sockets. This config overrides the SPARK_LOCAL_IP environment variable (see below). It also allows a different address from the local one to be advertised to executors or external systems. This is useful, for example, when running containers with bridged networking. For this to properly work, the different ports used by the driver (RPC, block manager and UI) need to be forwarded from the container's host.

`spark.driver.host`	(local hostname)	Hostname or IP address for the driver. 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.network.timeout`	120s	Default timeout for all network interactions. This config will be used in place of `spark.core.connection.ack.wait.timeout`, `spark.storage.blockManagerSlaveTimeoutMs`, `spark.shuffle.io.connectionTimeout`, `spark.rpc.askTimeout` or `spark.rpc.lookupTimeout` if they are not configured.
`spark.port.maxRetries`	16	Maximum number of retries when binding to a port before giving up. When a port is given a specific value (non 0), each subsequent retry will increment the port used in the previous attempt by 1 before retrying. This essentially allows it to try a range of ports from the start port specified to port + maxRetries.
`spark.rpc.numRetries`	3	Number of times to retry before an RPC task gives up. An RPC task will run at most times of this number.
`spark.rpc.retry.wait`	3s	Duration for an RPC ask operation to wait before retrying.
`spark.rpc.askTimeout`	`spark.network.timeout`	Duration for an RPC ask operation to wait before timing out.
`spark.rpc.lookupTimeout`	120s	Duration for an RPC remote endpoint lookup operation to wait before timing out.

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.locality.wait`	3s	How long 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`	30s	Maximum amount of time to wait for resources to register before scheduling begins.
`spark.scheduler.minRegisteredResourcesRatio`	0.8 for YARN mode; 0.0 for standalone mode and Mesos coarse-grained mode	The minimum ratio of registered resources (registered resources / total expected resources) (resources are executors in yarn mode, CPU cores in standalone mode and Mesos coarsed-grained mode ['spark.cores.max' value is total expected resources for Mesos coarse-grained 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`	1s	The interval length for the scheduler to revive the worker resource offers to run tasks.
`spark.blacklist.enabled`	false	If set to "true", prevent Spark from scheduling tasks on executors that have been blacklisted due to too many task failures. The blacklisting algorithm can be further controlled by the other "spark.blacklist" configuration options.
`spark.blacklist.timeout`	1h	(Experimental) How long a node or executor is blacklisted for the entire application, before it is unconditionally removed from the blacklist to attempt running new tasks.
`spark.blacklist.task.maxTaskAttemptsPerExecutor`	1	(Experimental) For a given task, how many times it can be retried on one executor before the executor is blacklisted for that task.
`spark.blacklist.task.maxTaskAttemptsPerNode`	2	(Experimental) For a given task, how many times it can be retried on one node, before the entire node is blacklisted for that task.
`spark.blacklist.stage.maxFailedTasksPerExecutor`	2	(Experimental) How many different tasks must fail on one executor, within one stage, before the executor is blacklisted for that stage.
`spark.blacklist.stage.maxFailedExecutorsPerNode`	2	(Experimental) How many different executors are marked as blacklisted for a given stage, before the entire node is marked as failed for the stage.
`spark.blacklist.application.maxFailedTasksPerExecutor`	2	(Experimental) How many different tasks must fail on one executor, in successful task sets, before the executor is blacklisted for the entire application. Blacklisted executors will be automatically added back to the pool of available resources after the timeout specified by `spark.blacklist.timeout`. Note that with dynamic allocation, though, the executors may get marked as idle and be reclaimed by the cluster manager.
`spark.blacklist.application.maxFailedExecutorsPerNode`	2	(Experimental) How many different executors must be blacklisted for the entire application, before the node is blacklisted for the entire application. Blacklisted nodes will be automatically added back to the pool of available resources after the timeout specified by `spark.blacklist.timeout`. Note that with dynamic allocation, though, the executors on the node may get marked as idle and be reclaimed by the cluster manager.
`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`	100ms	How often Spark will check for tasks to speculate.
`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	Fraction 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 failures of any particular task before giving up on the job. The total number of failures spread across different tasks will not cause the job to fail; a particular task has to fail this number of attempts. 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, which scales the number of executors registered with this application up and down based on the workload. For more detail, see the description here. This requires `spark.shuffle.service.enabled` to be set. The following configurations are also relevant: `spark.dynamicAllocation.minExecutors`, `spark.dynamicAllocation.maxExecutors`, and `spark.dynamicAllocation.initialExecutors`
`spark.dynamicAllocation.executorIdleTimeout`	60s	If dynamic allocation is enabled and an executor has been idle for more than this duration, the executor will be removed. For more detail, see this description.
`spark.dynamicAllocation.cachedExecutorIdleTimeout`	infinity	If dynamic allocation is enabled and an executor which has cached data blocks has been idle for more than this duration, the executor will be removed. For more details, see this description.
`spark.dynamicAllocation.initialExecutors`	`spark.dynamicAllocation.minExecutors`	Initial number of executors to run if dynamic allocation is enabled.
		If `--num-executors` (or `spark.executor.instances`) is set and larger than this value, it will be used as the initial number of executors.
`spark.dynamicAllocation.maxExecutors`	infinity	Upper bound for the number of executors if dynamic allocation is enabled.
`spark.dynamicAllocation.minExecutors`	0	Lower bound for the number of executors if dynamic allocation is enabled.
`spark.dynamicAllocation.schedulerBacklogTimeout`	1s	If dynamic allocation is enabled and there have been pending tasks backlogged for more than this duration, new executors will be requested. For more detail, see this description.
`spark.dynamicAllocation.sustainedSchedulerBacklogTimeout`	`schedulerBacklogTimeout`	Same as `spark.dynamicAllocation.schedulerBacklogTimeout`, but used only for subsequent executor requests. For more detail, see this description.

Security

Property Name	Default	Meaning
`spark.acls.enable`	false	Whether Spark acls should be enabled. If enabled, this see if the user has access permissions to view or modi Note this requires the user to be known, so if the user across as null no checks are done. Filters can be used to authenticate and set the user.
`spark.admin.acls`	Empty	Comma separated list of users/administrators that hav modify access to all Spark jobs. This can be used if yo shared cluster and have a set of administrators or devs debug when things do not work. Putting a "*" in the list user can have the privilege of admin.
`spark.admin.acls.groups`	Empty	Comma separated list of groups that have view and m to all Spark jobs. This can be used if you have a set of administrators or developers who help maintain and de underlying infrastructure. Putting a "*" in the list means any group can have the privilege of admin. The user gr obtained from the instance of the groups mapping prov specified by `spark.user.groups.mapping`. Check the e `spark.user.groups.mapping` for more details.
`spark.user.groups.mapping`	`org.apache.spark.security.ShellBasedGroupsMappingProvider`	The list of groups for a user are determined by a group service defined by the trait org.apache.spark.security.GroupMappingServiceProvid can configured by this property. A default unix shell ba

		implementation is provided `org.apache.spark.security.ShellBasedGroupsMappi` which can be specified to resolve a list of groups for a This implementation supports only a Unix/Linux based environment. Windows environment is currently **not** su However, a new platform/protocol can be supported b implementing the trait `org.apache.spark.security.GroupMappingServicePr`
`spark.authenticate`	false	Whether Spark authenticates its internal connections. `spark.authenticate.secret` if not running on YARN.
`spark.authenticate.secret`	None	Set the secret key used for Spark to authenticate betw components. This needs to be set if not running on YA authentication is enabled.
`spark.authenticate.enableSaslEncryption`	false	Enable encrypted communication when authentication This is supported by the block transfer service and the endpoints.
`spark.network.sasl.serverAlwaysEncrypt`	false	Disable unencrypted connections for services that sup authentication. This is currently supported by the exter service.
`spark.network.aes.enabled`	false	Enable AES for over-the-wire encryption. This is suppo RPC and the block transfer service. This option has pr over SASL-based encryption if both are enabled.
`spark.network.aes.keySize`	16	The bytes of AES cipher key which is effective when A enabled. AES works with 16, 24 and 32 bytes keys.
`spark.network.aes.config.*`	None	Configuration values for the commons-crypto library, s which cipher implementations to use. The config name the name of commons-crypto configuration without th "commons.crypto" prefix.
`spark.core.connection.ack.wait.timeout`	`spark.network.timeout`	How long for the connection to wait for ack to occur b out and giving up. To avoid unwilling timeout caused b pause like GC, you can set larger value.
`spark.core.connection.auth.wait.timeout`	30s	How long for the connection to wait for authentication before timing out and giving up.
`spark.modify.acls`	Empty	Comma separated list of users that have modify acces Spark job. By default only the user that started the Spa access to modify it (kill it for example). Putting a "*" in means any user can have access to modify it.
`spark.modify.acls.groups`	Empty	Comma separated list of groups that have modify acce Spark job. This can be used if you have a set of admin developers from the same team to have access to con Putting a "*" in the list means any user in any group ha to modify the Spark job. The user groups are obtained instance of the groups mapping provider specified by `spark.user.groups.mapping`. Check the entry `spark.user.groups.mapping` for more details.
`spark.ui.filters`	None	Comma separated list of filter class names to apply to web UI. The filter should be a standard javax servlet Fi Parameters to each filter can also be specified by setti 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,para`
`spark.ui.view.acls`	Empty	Comma separated list of users that have view access t web ui. By default only the user that started the Spark access. Putting a "*" in the list means any user can hav access to this Spark job.
`spark.ui.view.acls.groups`	Empty	Comma separated list of groups that have view access Spark web ui to view the Spark Job details. This can b

you have a set of administrators or developers or users
monitor the Spark job submitted. Putting a "*" in the lis
user in any group can view the Spark job details on the
ui. The user groups are obtained from the instance of t
mapping provider specified by `spark.user.groups.ma`
Check the entry `spark.user.groups.mapping` for more

TLS / SSL

Property Name	Default	Meaning
`spark.ssl.enabled`	false	Whether to enable SSL connections on all supported protocols. When `spark.ssl.enabled` is configured, `spark.ssl.protocol` is required. 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`. Example values for `YYY` include `fs`, `ui`, `standalone`, and `historyServer`. See SSL Configuration for details on hierarchical SSL configuration for services.
`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. Note: If not set, it will use the default cipher suites of JVM.
`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.keyStoreType`	JKS	The type of 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.needClientAuth`	false	Set true if SSL needs client authentication.
`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.ssl.trustStoreType`	JKS	The type of the trust-store.

Spark SQL

Running the `SET -v` command will show the entire list of the SQL configuration.

Scala Java Python R

Scala Java Python R

```scala
// spark is an existing SparkSession
spark.sql("SET -v").show(numRows = 200, truncate = false)
```

Spark Streaming

Property Name	Default	Meaning
`spark.streaming.backpressure.enabled`	false	Enables or disables Spark Streaming's internal backpressure mechanism (since 1.5). This enables the Spark Streaming to control the receiving rate based on the current batch scheduling delays and processing times so that the system receives only as fast as the system can process. Internally, this dynamically sets the maximum receiving rate of receivers. This rate is upper bounded by the values

		`spark.streaming.receiver.maxRate` and `spark.streaming.kafka.maxRatePerPartition` if they are set (see below).
`spark.streaming.backpressure.initialRate`	not set	This is the initial maximum receiving rate at which each receiver will receive data for the first batch when the backpressure mechanism is enabled.
`spark.streaming.blockInterval`	200ms	Interval 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.stopGracefullyOnShutdown`	false	If `true`, Spark shuts down the `StreamingContext` gracefully on JVM shutdown rather than immediately.
`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.
`spark.streaming.kafka.maxRetries`	1	Maximum number of consecutive retries the driver will make in order to find the latest offsets on the leader of each partition (a default value of 1 means that the driver will make a maximum of 2 attempts). Only applies to the new Kafka direct stream API.
`spark.streaming.ui.retainedBatches`	1000	How many batches the Spark Streaming UI and status APIs remember before garbage collecting.
`spark.streaming.driver.writeAheadLog.closeFileAfterWrite`	false	Whether to close the file after writing a write ahead log record on the driver. Set this to 'true' when you want to use S3 (or any file system that does not support flushing) for the metadata WAL on the driver.
`spark.streaming.receiver.writeAheadLog.closeFileAfterWrite`	false	Whether to close the file after writing a write ahead log record on the receivers. Set this to 'true' when you want to use S3 (or any file system that does not support flushing) for the data WAL on the receivers.

SparkR

Property Name	Default	Meaning
`spark.r.numRBackendThreads`	2	Number of threads used by RBackend to handle RPC calls from SparkR package.
`spark.r.command`	Rscript	Executable for executing R scripts in cluster modes for both driver and workers.
`spark.r.driver.command`	spark.r.command	Executable for executing R scripts in client modes for driver. Ignored in cluster modes.
`spark.r.shell.command`	R	Executable for executing sparkR shell in client modes for driver. Ignored in cluster modes. It is the same as environment variable `SPARKR_DRIVER_R`, but take

		precedence over it. `spark.r.shell.command` is used for sparkR shell while `spark.r.driver.command` is used for running R script.
`spark.r.backendConnectionTimeout`	6000	Connection timeout set by R process on its connection to RBackend in seconds.
`spark.r.heartBeatInterval`	100	Interval for heartbeats sent from SparkR backend to R process to prevent connection timeout.

Deploy

Property Name	Default	Meaning
`spark.deploy.recoveryMode`	NONE	The recovery mode setting to recover submitted Spark jobs with cluster mode when it failed and relaunches. This is only applicable for cluster mode when running with Standalone or Mesos.
`spark.deploy.zookeeper.url`	None	When `spark.deploy.recoveryMode` is set to ZOOKEEPER, this configuration is used to set the zookeeper URL to connect to.
`spark.deploy.zookeeper.dir`	None	When `spark.deploy.recoveryMode` is set to ZOOKEEPER, this configuration is used to set the zookeeper directory to store recovery state.

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 in both driver and workers (default is `python2.7` if available, otherwise `python`). Property `spark.pyspark.python` take precedence if it is set
`PYSPARK_DRIVER_PYTHON`	Python binary executable to use for PySpark in driver only (default is `PYSPARK_PYTHON`). Property `spark.pyspark.driver.python` take precedence if it is set
`SPARKR_DRIVER_R`	R binary executable to use for SparkR shell (default is `R`). Property `spark.r.shell.command` take precedence if it is set
`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.

Note: When running Spark on YARN in `cluster` mode, environment variables need to be set using the `spark.yarn.appMasterEnv.[EnvironmentVariableName]` property in your `conf/spark-defaults.conf` file. Environment variables that are set in `spark-env.sh` will not be reflected in the YARN Application Master process in `cluster` mode. See the YARN-related Spark Properties for more information.

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 configuration files (spark-defaults.conf, spark-env.sh, log4j.properties, etc) from this directory.

Inheriting Hadoop Cluster Configuration

If you plan to read and write from HDFS using Spark, there are two Hadoop configuration files that should be included on Spark’s classpath:

- `hdfs-site.xml`, which provides default behaviors for the HDFS client.
- `core-site.xml`, which sets the default filesystem name.

The location of these configuration files varies across CDH and HDP versions, but a common location is inside of `/etc/hadoop/conf`. Some tools, such as Cloudera Manager, create configurations on-the-fly, but offer a mechanisms to download copies of them.

To make these files visible to Spark, set `HADOOP_CONF_DIR` in `$SPARK_HOME/spark-env.sh` to a location containing the configuration files.