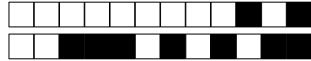




## Transformations

The following table lists some of the common transformations supported by Spark. Refer to the RDD API doc ([Scala](#), [Java](#), [Python](#)) and pair RDD functions doc ([Scala](#), [Java](#)) for details.

Transformation	Meaning
<code>map(func)</code>	Return a new distributed dataset formed by passing each element of the source through a function <i>func</i> .
<code>filter(func)</code>	Return a new dataset formed by selecting those elements of the source on which <i>func</i> returns true.
<code>flatMap(func)</code>	Similar to map, but each input item can be mapped to 0 or more output items (so <i>func</i> should return a Seq rather than a single item).
<code>mapPartitions(func)</code>	Similar to map, but runs separately on each partition (block) of the RDD, so <i>func</i> must be of type <code>Iterator&lt;T&gt; =&gt; Iterator&lt;U&gt;</code> when running on an RDD of type T.
<code>mapPartitionsWithIndex(func)</code>	Similar to mapPartitions, but also provides <i>func</i> with an integer value representing the index of the partition, so <i>func</i> must be of type <code>(Int, Iterator&lt;T&gt;) =&gt; Iterator&lt;U&gt;</code> when running on an RDD of type T.
<code>sample(withReplacement, fraction, seed)</code>	Sample a fraction <i>fraction</i> of the data, with or without replacement, using a given random number generator seed.
<code>union(otherDataset)</code>	Return a new dataset that contains the union of the elements in the source dataset and the argument.
<code>intersection(otherDataset)</code>	Return a new RDD that contains the intersection of elements in the source dataset and the argument.
<code>distinct([numTasks])</code>	Return a new dataset that contains the distinct elements of the source dataset.
<code>groupByKey([numTasks])</code>	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. <b>Note:</b> If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using <code>reduceByKey</code> or <code>combineByKey</code> will yield much better performance. <b>Note:</b> By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional <code>numTasks</code> argument to set a different number of tasks.
<code>reduceByKey(func, [numTasks])</code>	When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function <i>func</i> , which must be of type <code>(V,V) =&gt; V</code> . Like in <code>groupByKey</code> , the number of reduce tasks is configurable through an optional second argument.
<code>aggregateByKey(zeroValue)(seqOp, combOp, [numTasks])</code>	When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations. Like in <code>groupByKey</code> , the number of reduce tasks is configurable through an optional second argument.
<code>sortByKey([ascending], [numTasks])</code>	When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean <i>ascending</i> argument.
<code>join(otherDataset, [numTasks])</code>	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through <code>leftOuterJoin</code> , <code>rightOuterJoin</code> , and <code>fullOuterJoin</code> .
<code>cogroup(otherDataset, [numTasks])</code>	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, Iterable<V>, Iterable<W>) tuples. This operation is also called <code>groupWith</code> .
<code>cartesian(otherDataset)</code>	When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements).
<code>pipe(command, [envVars])</code>	Pipe each partition of the RDD through a shell command, e.g. a Perl or bash script. RDD elements are written to the process's stdin and lines output to its stdout are returned as an RDD of strings.
<code>coalesce(numPartitions)</code>	Decrease the number of partitions in the RDD to <code>numPartitions</code> . Useful for running operations more efficiently after filtering down a large dataset.
<code>repartition(numPartitions)</code>	Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network.
<code>repartitionAndSortWithinPartitions(partitioner)</code>	Repartition the RDD according to the given partitioner and, within each resulting partition, sort records by their keys. This is more efficient than calling <code>repartition</code> and then sorting within each partition because it can push the sorting down into the shuffle machinery.



## Actions

The following table lists some of the common actions supported by Spark. Refer to the RDD API doc ([Scala](#), [Java](#), [Python](#)) and pair RDD functions doc ([Scala](#), [Java](#)) for details.

Action	Meaning
<b>reduce</b> ( <i>func</i> )	Aggregate the elements of the dataset using a function <i>func</i> (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
<b>collect</b> ()	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
<b>count</b> ()	Return the number of elements in the dataset.
<b>first</b> ()	Return the first element of the dataset (similar to <code>take(1)</code> ).
<b>take</b> ( <i>n</i> )	Return an array with the first <i>n</i> elements of the dataset. Note that this is currently not executed in parallel. Instead, the driver program computes all the elements.
<b>takeSample</b> ( <i>withReplacement</i> , <i>num</i> , [ <i>seed</i> ])	Return an array with a random sample of <i>num</i> elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.
<b>takeOrdered</b> ( <i>n</i> , [ <i>ordering</i> ])	Return the first <i>n</i> elements of the RDD using either their natural order or a custom comparator.
<b>saveAsTextFile</b> ( <i>path</i> )	Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call <code>toString</code> on each element to convert it to a line of text in the file.
<b>saveAsSequenceFile</b> ( <i>path</i> ) (Java and Scala)	Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. This is available on RDDs of key-value pairs that either implement Hadoop's <code>Writable</code> interface. In Scala, it is also available on types that are implicitly convertible to <code>Writable</code> (Spark includes conversions for basic types like <code>Int</code> , <code>Double</code> , <code>String</code> , etc).
<b>saveAsObjectFile</b> ( <i>path</i> ) (Java and Scala)	Write the elements of the dataset in a simple format using Java serialization, which can then be loaded using <code>SparkContext.objectFile()</code> .
<b>countByKey</b> ()	Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.
<b>foreach</b> ( <i>func</i> )	Run a function <i>func</i> on each element of the dataset. This is usually done for side effects such as updating an accumulator variable (see below) or interacting with external storage systems.



## RDD Persistence

One of the most important capabilities in Spark is *persisting* (or *caching*) a dataset in memory across operations. When you persist an RDD, each node stores any partitions of it that it computes in memory and reuses them in other actions on that dataset (or datasets derived from it). This allows future actions to be much faster (often by more than 10x). Caching is a key tool for iterative algorithms and fast interactive use.

You can mark an RDD to be persisted using the `persist()` or `cache()` methods on it. The first time it is computed in an action, it will be kept in memory on the nodes. Spark's cache is fault-tolerant – if any partition of an RDD is lost, it will automatically be recomputed using the transformations that originally created it.

In addition, each persisted RDD can be stored using a different *storage level*, allowing you, for example, to persist the dataset on disk, persist it in memory but as serialized Java objects (to save space), replicate it across nodes, or store it off-heap in [Tachyon](#). These levels are set by passing a `StorageLevel` object ([Scala](#), [Java](#), [Python](#)) to `persist()`. The `cache()` method is a shorthand for using the default storage level, which is `StorageLevel.MEMORY_ONLY` (store deserialized objects in memory). The full set of storage levels is:

Storage Level	Meaning
MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER	Store RDD as <i>serialized</i> Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a <a href="#">fast serializer</a> , but more CPU-intensive to read.
MEMORY_AND_DISK_SER	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	Same as the levels above, but replicate each partition on two cluster nodes.
OFF_HEAP (experimental)	Store RDD in serialized format in <a href="#">Tachyon</a> . Compared to MEMORY_ONLY_SER, OFF_HEAP reduces garbage collection overhead and allows executors to be smaller and to share a pool of memory, making it attractive in environments with large heaps or multiple concurrent applications. Furthermore, as the RDDs reside in Tachyon, the crash of an executor does not lead to losing the in-memory cache. In this mode, the memory in Tachyon is discardable. Thus, Tachyon does not attempt to reconstruct a block that it evicts from memory.

**Note:** In Python, stored objects will always be serialized with the [Pickle](#) library, so it does not matter whether you choose a serialized level.

Spark also automatically persists some intermediate data in shuffle operations (e.g. `reduceByKey`), even without users calling `persist`. This is done to avoid recomputing the entire input if a node fails during the shuffle. We still recommend users call `persist` on the resulting RDD if they plan to reuse it.