Spark is lazy, so nothing will be executed unless you call some transformation or action that will trigger job creation and execution. Look at the following snippet of the word-count example.

Therefore, RDD transformation is not a set of data but is a step in a program (might be the only step) telling Spark how to get data and what to do with it.

Given below is a list of RDD transformations.

S. No	Transformations & Meaning
1	**map(func)** Returns a new distributed dataset, formed by passing each element of the source through a function **func**.
2	**filter(func)** Returns a new dataset formed by selecting those elements of the source on which **func** returns true.
3	**flatMap(func)** Similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item).
4	**mapPartitions(func)** Similar to map, but runs separately on each partition (block) of the RDD, so **func** must be of type Iterator<T> => Iterator<U> when running on an RDD of type T.
5	**mapPartitionsWithIndex(func)** Similar to map Partitions, but also provides **func** with an integer value representing the index of the partition, so **func** must be of type (Int, Iterator<T>) => Iterator<U> when running on an RDD of type T.
6	**sample(withReplacement, fraction, seed)** Sample a **fraction** of the data, with or without replacement, using a given random number generator seed.
7	**union(otherDataset)** Returns a new dataset that contains the union of the elements in the source

	dataset and the argument.
8	**intersection(otherDataset)** Returns a new RDD that contains the intersection of elements in the source dataset and the argument.
9	**distinct([numTasks]))** Returns a new dataset that contains the distinct elements of the source dataset.
10	**groupByKey([numTasks])** When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. **Note:** If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using reduceByKey or aggregateByKey will yield much better performance.
11	**reduceByKey(func, [numTasks])** 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 *func*, which must be of type (V, V) => V. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.
12	**aggregateByKey(zeroValue)(seqOp, combOp, [numTasks])** 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 from the input value type, while avoiding unnecessary allocations. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.
13	**sortByKey([ascending], [numTasks])** 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 ascending argument.
14	**join(otherDataset, [numTasks])**

	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 leftOuterJoin, rightOuterJoin, and fullOuterJoin.
15	**cogroup(otherDataset, [numTasks])** 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 group With.
16	**cartesian(otherDataset)** When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements).
17	**pipe(command, [envVars])** 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.
18	**coalesce(numPartitions)** Decrease the number of partitions in the RDD to numPartitions. Useful for running operations more efficiently after filtering down a large dataset.
19	**repartition(numPartitions)** 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.
20	**repartitionAndSortWithinPartitions(partitioner)** Repartition the RDD according to the given partitioner and, within each resulting partition, sort records by their keys. This is more efficient than calling repartition and then sorting within each partition because it can push the sorting down into the shuffle machinery.

Actions

The following table gives a list of Actions, which return values.

S.No	Action & Meaning
1	**reduce(func)** Aggregate the elements of the dataset using a function **func** (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
2	**collect**() Returns 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.
3	**count**() Returns the number of elements in the dataset.
4	**first**() Returns the first element of the dataset (similar to take (1)).
5	**take**(*n*) Returns an array with the first ***n*** elements of the dataset.
6	**takeSample (withReplacement,num, [seed])** Returns an array with a random sample of **num** elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.
7	**takeOrdered(n, [ordering])** Returns the first **n** elements of the RDD using either their natural order or a custom comparator.
8	**saveAsTextFile(path)** Writes 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 calls toString on each element to convert it to a line of text

	in the file.
9	**saveAsSequenceFile(path) (Java and Scala)** Writes 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 implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc).
10	**saveAsObjectFile(path) (Java and Scala)** Writes the elements of the dataset in a simple format using Java serialization, which can then be loaded using SparkContext.objectFile().
11	**countByKey**() Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.
12	**foreach(func)** Runs a function **func** on each element of the dataset. This is usually, done for side effects such as updating an Accumulator or interacting with external storage systems. **Note**: modifying variables other than Accumulators outside of the foreach() may result in undefined behavior. See Understanding closures for more details.

Programming with RDD

Let us see the implementations of few RDD transformations and actions in RDD programming with the help of an example.

Example

Consider a word count example: It counts each word appearing in a document. Consider the following text as an input and is saved as an **input.txt** file in a home directory.

input.txt: input file.

```
people are not as beautiful as they look,
as they walk or as they talk.
they are only as beautiful  as they love,
```