API for SparkContext	Meaning
new **SparkContext**(config: SparkConf)	Create a SparkContext that loads settings from a Spark Config object describing the application configuration. Any settings in this config overrides the default configs as well as system properties.
new **SparkContext**(master: String, appName: String, conf: SparkConf)	Alternative constructor that allows setting common Spark properties directly, : Cluster URL to connect to (e.g. mesos://host:port, spark://host:port, local[4]), a name for your application, to display on the cluster web UI and a SparkConf object specifying other Spark parameters
textFile(path: String, minPartitions: Int = defaultMinPartitions): RDD[String]	Read a text file from HDFS, a local file system (available on all nodes), or any Hadoop-supported file system URI, and return it as an RDD of Strings. Path is the path to the text file on a supported file system, minPartitions is the suggested minimum number of partitions for the resulting RDD. It returns RDD of lines of the text file
sequenceFile[K, V](path: String, keyClass: Class[K], valueClass: Class[V]): RDD[(K, V)]	Get an RDD for a Hadoop SequenceFile with given key and value types. Path is the directory to the input data files, the path can be comma separated paths as a list of inputs, keyClass (resp. valueClass) is the `Class` of the key (resp. the value) associated with `SequenceFileInputFormat`. It returns RDD of tuples of key and corresponding value.
parallelize[T](seq: Seq[T], numSlices: Int): RDD[T]	Distribute a local Scala collection to form an RDD. NumSlices is number of partitions to divide the collection into. It returns the RDD representing distributed collection
newAPIHadoopFile[K, V, F <: InputFormat[K, V]]path: String, fClass: Class[F], kClass: Class[K], vClass: Class[V], conf: Configuration = hadoopConfiguration): RDD[(K, V)]	Get an RDD for a given Hadoop file with an arbitrary new API InputFormat and extra configuration options to pass to the input format. Path is the directory to the input data files, the path can be comma separated paths as a list of inputs, fclass is the storage format of the data to be read , kClass (resp. vClass) is the `Class` of the key (resp. the value) Hadoop configuration. It returns RDD of tuples of key and corresponding value
broadcast[T](value: T): Broadcast[T]	Broadcast a read-only variable to the cluster, returning a org.apache.spark.broadcast.Broadcast object for reading it in distributed functions. The variable will be sent to each cluster only once.
collectionAccumulator[T]: CollectionAccumulator[T]	Create and register a `CollectionAccumulator`, which starts with empty list and accumulates inputs by adding them into the list.
doubleAccumulator: DoubleAccumulator **longAccumulator**: LongAccumulator	Create and register a double or long accumulator, which starts with 0 and accumulates inputs by `add`.
register(acc: AccumulatorV2[_, _]): Unit	Register the given accumulator. Accumulators must be registered before use, or it will throw exception.
stop(): Unit	Shut down the SparkContext.

Simple transformation for RDD[T]	Meaning
map[U](f: (T) =>U): RDD[U]	Return a new distributed dataset formed by passing each element of the source through a function *f* .
mapValues[U](f: V => U): RDD[(K, U)]	Pass each value in the key-value pair RDD through a map function without changing the keys this also retains the original RDD's partitioning.
filter(f: (T) =>Boolean): RDD[T]	Return a new dataset formed by selecting those elements of the source on which *f* returns true.
flatMap[U](f: (T) =>TraversableOnce[U]): RDD[U]	Similar to map, but each input item can be mapped to 0 or more output items (so *f* should return a Seq rather than a single item).
flatMapValues[U](f: V => TraversableOnce[U]): RDD[(K, U)]	Pass each value in the key-value pair RDD through a flatMap function without changing the keys ; this also retains the original RDD's partitioning.

union(other: RDD[T]): RDD[T]	Return a new dataset that contains the union of the elements in the source dataset and the argument.
zip[U](other: RDD[U]): RDD[(T, U)]	Zips this RDD with another one, returning key-value pairs with the first element in each RDD, second element in each RDD, etc. *Assumes that the two RDDs have the same number of partitions and the same number of elements in each corresponding partition* (e.g. one was made through a map on the other).
zipWithIndex(): RDD[(T, Long)]	Zips this RDD with its element indices. The ordering is first based on the partition index and then the ordering of items within each partition. So the first item in the first partition gets index 0, and the last item in the last partition receives the largest index. This is similar to Scala's zipWithIndex but it uses Long instead of Int as the index type. This method needs to trigger a spark job when this RDD contains more than one partitions.
keyBy[K](f: (T) =>K): RDD[(K, T)]	Creates tuples of the elements in this RDD by applying f
glom(): RDD[Array[T]]	Return an RDD created by coalescing all elements within each partition into an array.

Large transformation for RDD[T]	Meaning
intersection(other: RDD[T]): RDD[T] **intersection**(other: RDD[T], numPartitions: Int): RDD[T] **intersection**(other: RDD[T], part: Partitioner): RDD[T]	Return the intersection of this RDD and another one. The output will not contain any duplicate elements, even if the input RDDs did.
distinct(): RDD[T] **distinct**(numPartitions: Int): RDD[T]	Return a new dataset that contains the distinct elements of the source dataset.
cartesian[U](other: RDD[U]): RDD[(T, U)]	When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements).
repartition(numPartitions: Int): RDD[T]	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.
subtract(other: RDD[T]): RDD[T] **subtract**(other: RDD[T], numPartitions: Int): RDD[T] **subtract**(other: RDD[T], p: Partitioner): RDD[T]	Return an RDD with the elements from *this* that are not in *other*.
sortBy[K](f: (T) =>K, ascending: Boolean = true , numPartitions: Int = this.partitions.length): RDD[T]	Return this RDD sorted by the given key function.

Large transformation for RDD[T] and T= (K,V)	Meaning
groupByKey(): RDD[(K, Iterable[V])] **groupByKey**(numPartitions:Int): RDD[(K, Iterable[V])] **groupByKey**(p:Partitioner): RDD[(K, Iterable[V])]	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs.
reduceByKey(func: (V, V) =>V): RDD[(K, V)] **reduceByKey**(numPartitions:Int, func: (V, V) =>V): RDD[(K, V)] **reduceByKey**(partitioner: Partitioner, func: (V, V) =>V): RDD[(K, V)]	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.
sortByKey(ascending: Boolean = true, numPartitions: Int = self.partitions.length) : RDD[(K, V)]	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.

join[W](other: RDD[(K, W)]): RDD[(K, (V, W))] **join**[W](other: RDD[(K, W)], numPartitions: Int): RDD[(K, (V, W))] **join**[W](other: RDD[(K, W)], p: Partitioner): RDD[(K, (V, W))]	Return an RDD containing all pairs of elements with matching keys in *this* and *other*. Each pair of elements will be returned as a (k, (v1, v2)) tuple, where (k, v1) is in *this* and (k, v2) is in *other*
leftOuterJoin[W](other: RDD[(K, W)]): RDD[(K, (V, Option[W]))] **leftOuterJoin**[W](other: RDD[(K, W)], numPartitions: Int): RDD[(K, (V, Option[W]))] **leftOuterJoin**[W](other: RDD[(K, W)], partitioner: Partitioner): RDD[(K, (V, Option[W]))]	Perform a left outer join of `this` and `other`. For each element (k, v) in `this`, the resulting RDD will either contain all pairs (k, (v, Some(w))) for w in `other`, or the pair (k, (v, None)) if no elements in `other` have key k.
rightOuterJoin[W](other: RDD[(K, W)]): RDD[(K, (Option[V], W))] **rightOuterJoin**[W](other: RDD[(K, W)], numPartitions: Int): RDD[(K, (Option[V], W))] **rightOuterJoin**[W](other: RDD[(K, W)], partitioner: Partitioner): RDD[(K, (Option[V], W))]	Perform a right outer join of `this` and `other`. For each element (k, w) in `other`, the resulting RDD will either contain all pairs (k, (Some(v), w)) for v in `this`, or the pair (k, (None, w)) if no elements in `this` have key k.
fullOuterJoin[W](other: RDD[(K, W)]): RDD[(K, (Option[V], Option[W]))] **fullOuterJoin**[W](other: RDD[(K, W)], numPartitions: Int): RDD[(K, (Option[V], Option[W]))] **fullOuterJoin**[W](other: RDD[(K, W)], partitioner: Partitioner): RDD[(K, (Option[V], Option[W]))]	Perform a full outer join of `this` and `other`. For each element (k, v) in `this`, the resulting RDD will either contain all pairs (k, (Some(v), Some(w))) for w in `other`, or the pair (k, (Some(v), None)) if no elements in `other` have key k. Similarly, for each element (k, w) in `other`, the resulting RDD will either contain all pairs (k, (Some(v), Some(w))) for v in `this`, or the pair (k, (None, Some(w))) if no elements in `this` have key k.
cogroup[W](o: RDD[(K, W)]): RDD[(K, (Iterable[V], Iterable[W]))] **cogroup**[W](o: RDD[(K, W)], numPart: Int): RDD[(K, (Iterable[V], Iterable[W]))] **cogroup**[W](o: RDD[(K, W)], p: Partitioner): RDD[(K, (Iterable[V], Iterable[W]))]	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 `groupWith`.
subtractByKey[W](other: RDD[(K, W)]): RDD[(K, V)] **subtractByKey**[W](other: RDD[(K, W)], numPartitions: Int): RDD[(K, V)] **subtractByKey**[W](other: RDD[(K, W)], p: Partitioner): RDD[(K, V)]	Return an RDD with the pairs from *this* whose keys are not in *other*

Action for RDD[T]	Meaning
reduce(f: (T, T) =>T): T	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.
collect(): Array[T]	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.
count():Long	Return the number of elements in the dataset.
first(): T	Return the first element of the dataset (similar to take(1)).
take(num: Int): Array[T]	Return an array with the first *n* elements of the dataset.
top(num: Int)(implicit ord: Ordering[T]): Array[T]	Returns the top k (largest) elements from this RDD as defined by the specified implicit Ordering[T] and maintains the ordering.
saveAsObjectFile(path: String): Unit	Save this RDD as a SequenceFile of serialized objects.
saveAsTextFile(path: String): Unit	Save this RDD as a text file, using string representations of elements.
foreach(f: (T) =>Unit): Unit	Run 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.