



## Zhen He

Associate Professor

Department of Computer Science and Computer Engineering  
La Trobe University  
Bundoora, Victoria 3086  
Australia

Tel : + 61 3 9479 3036  
Email: z.he@latrobe.edu.au

Building: Beth Gleeson, Room: 235

[Home](#)

### RDD function calls

[aggregate](#)

[aggregateByKey \[Pair\]](#)

[cartesian](#)

[checkpoint](#)

[coalesce, repartition](#)

[cogroup \[pair\], groupWith \[Pair\]](#)

[collect, toArray](#)

[collectAsMap \[pair\]](#)

[combineByKey \[pair\]](#)

[compute](#)

[context, sparkContext](#)

[count](#)

[countApprox](#)

[countApproxDistinct](#)

[countApproxDistinctByKey \[pair\]](#)

[countByKey \[pair\]](#)

[countByKeyApprox \[pair\]](#)

[countByValue](#)

[countByValueApprox](#)

[dependencies](#)

[distinct](#)

[first](#)

[filter](#)

[filterByRange \[Ordered\]](#)

[filterWith](#)

[flatMap](#)

[flatMapValues \[Pair\]](#)

[flatMapWith](#)

[fold](#)

[foldByKey \[Pair\]](#)

[foreach](#)

[foreachPartition](#)

[foreachWith](#)

Our research group has a very strong focus on using and improving Apache Spark to solve real world programs. In order to use the RDD API and write examples to test their functionality. This has been a very useful exercise and we would like to share the experience.

Authors of examples: Matthias Langer and Zhen He  
Emails addresses: m.langer@latrobe.edu.au, z.he@latrobe.edu.au

These examples have only been tested for Spark version 1.4. We assume the functionality of Spark is stable and therefore we would like to share the experience.

If you find any errors in the example we would love to hear about them so we can fix them up. So please email us to let us know.

### The RDD API By Example

RDD is short for Resilient Distributed Dataset. RDDs are the workhorse of the Spark system. As a user, one can consider an RDD as a collection of partitions that can be processed in parallel.

However, an RDD is actually more than that. On cluster installations, separate data partitions can be on separate nodes. If an entire RDD is lost, the system is able to reconstruct the data of lost partitions by using lineage information. Lineage information is the record of the operations that created the RDD.

All RDDs available in Spark derive either directly or indirectly from the class RDD. This class comes with a large set of methods to create RDDs from various sources. These implementations have to overwrite some core functions to make the RDD be able to be serialized and deserialized.

One reason why Spark has lately become a very popular system for processing big data is that it does not impose restrictions on the core API of RDDs common enough to handle arbitrary data-types, many convenience functions are missing.

The basic RDD API considers each data item as a single value. However, users often want to work with key-value pairs. There are four extensions to the RDD API available in spark. They are as follows:

#### DoubleRDDFunctions

This extension contains many useful methods for aggregating numeric values. They become available if the data items are numeric.

#### PairRDDFunctions

Methods defined in this interface extension become available when the data items have a two component tuple structure.

#### OrderedRDDFunctions

Methods defined in this interface extension become available if the data items are two-component tuples where the second component is ordered.

#### SequenceFileRDDFunctions

This extension contains several methods that allow users to create Hadoop sequencefiles from RDDs. The data items must be of type Writable.

Since Spark will make methods with extended functionality automatically available to users when the data items fulfill the requirements, the function-name to indicate it belongs to an extension that requires the data items to conform to a certain format or type.

[Double] - Double RDD Functions

[Ordered] - OrderedRDDFunctions

[Pair] - PairRDDFunctions

[SeqFile] - SequenceFileRDDFunctions

### aggregate

The **aggregate** function allows the user to apply **two** different reduce functions to the RDD. The first reduce function is used to reduce the results of all partitions together to arrive at one final result. The ability to have two separate reduce functions for intra-partition function. The user also specifies an initial value. Here are some important facts.

[fullOuterJoin \[Pair\]](#)[generator.setGenerator](#)[getCheckpointFile](#)[preferredLocations](#)[getStorageLevel](#)[glom](#)[groupBy](#)[groupByKey \[Pair\]](#)[histogram \[Double\]](#)[id](#)[intersection](#)[isCheckpointed](#)[iterator](#)[join \[pair\]](#)[keyBy](#)[keys \[pair\]](#)[leftOuterJoin \[pair\]](#)[lookup \[pair\]](#)[map](#)[mapPartitions](#)[mapPartitionsWithContext](#)[mapPartitionsWithIndex](#)[mapPartitionsWithSplit](#)[mapValues \[pair\]](#)[mapWith](#)[max](#)[mean \[Double\].meanApprox \[Double\]](#)[min](#)[name.setName](#)[partitionBy \[Pair\]](#)[partitioner](#)[partitions](#)[persist.cache](#)[pipe](#)[randomSplit](#)[reduce](#)[reduceByKey \[Pair\]](#)[reduceByKeyLocally \[Pair\]](#)[reduceByKeyToDriver \[Pair\]](#)[repartition](#)[repartitionAndSortWithPartitions \[Ordered\]](#)[rightOuterJoin \[Pair\]](#)[sample](#)[sampleByKey \[Pair\]](#)[sampleByKeyExact \[Pair\]](#)

- The initial value is applied at both levels of reduce. So both at the intra partition reduction and across partition reduction.
- Both reduce functions have to be commutative and associative.
- Do not assume any execution order for either partition computations or combining partitions.
- Why would one want to use two input data types? Let us assume we do an archaeological site survey using a meta-image of a map that highlights these locations using the **aggregate** function. In this case the **zeroValue** could be a GPS coordinates to map coordinates and put a marker on the map at the respective position. **combOp (second reduce)**

### Listing Variants

```
def aggregate[U: ClassTag](zeroValue: U)(seqOp: (U, T) => U, combOp: (U, U) => U): U
```

### Examples 1

```
val z = sc.parallelize(List(1,2,3,4,5,6), 2)

// lets first print out the contents of the RDD with partition labels
def myfunc(index: Int, iter: Iterator[Int]) : Iterator[String] = {
  iter.toList.map(x => "[partID:" + index + ", val: " + x + "]").iterator
}

z.mapPartitionsWithIndex(myfunc).collect
res28: Array[String] = Array([partID:0, val: 1], [partID:0, val: 2], [partID:0, val: 3],
[partID:1, val: 4], [partID:1, val: 5], [partID:1, val: 6])

z.aggregate(0)(math.max(_,_), _+_ )
res40: Int = 9

// This example returns 16 since the initial value is 5
// reduce of partition 0 will be max(5, 1, 2, 3) = 5
// reduce of partition 1 will be max(5, 4, 5, 6) = 6
// final reduce across partitions will be 5 + 5 + 6 = 16
// note the final reduce include the initial value
z.aggregate(5)(math.max(_,_), _+_ )
res29: Int = 16

val z = sc.parallelize(List("a","b","c","d","e","f"),2)

//lets first print out the contents of the RDD with partition labels
def myfunc(index: Int, iter: Iterator[String]) : Iterator[String] = {
  iter.toList.map(x => "[partID:" + index + ", val: " + x + "]").iterator
}

z.mapPartitionsWithIndex(myfunc).collect
res31: Array[String] = Array([partID:0, val: a], [partID:0, val: b], [partID:0, val: c],
[partID:1, val: d], [partID:1, val: e], [partID:1, val: f])

z.aggregate(")(_+_+_+_ )
res115: String = abcdef

// See here how the initial value "x" is applied three times.
// - once for each partition
// - once when combining all the partitions in the second reduce function.
z.aggregate("x")(_+_+_+_ )
res116: String = xxdefxabc

// Below are some more advanced examples. Some are quite tricky to work out.

val z = sc.parallelize(List("12","23","345","4567"),2)
z.aggregate("")(x,y) => math.max(x.length, y.length).toString, (x,y) => x + y
res141: String = 42

z.aggregate("")(x,y) => math.min(x.length, y.length).toString, (x,y) => x + y
res142: String = 11

val z = sc.parallelize(List("12","23","345",""),2)
z.aggregate("")(x,y) => math.min(x.length, y.length).toString, (x,y) => x + y
res143: String = 10
```

The main issue with the code above is that the result of the inner **min** is a string of length 1. The zero in the output is due to the empty string being the last string in the list. We see this result because we are not reducing the empty string.

### Examples 2

```
val z = sc.parallelize(List("12","23","","345"),2)
z.aggregate("")(x,y) => math.min(x.length, y.length).toString, (x,y) => x + y
res144: String = 11
```

In contrast to the previous example, this example has the empty string at the beginning of the second partition. This result is dependent on the order of the data inside the partitions.)

[saveAsHadoopFile \[Pair\]](#),  
[saveAsHadoopDataset \[Pair\]](#),  
[saveAsNewAPIHadoopFile \[Pair\]](#)

[saveAsObjectFile](#)

[saveAsSequenceFile \[SeqFile\]](#)

[saveAsTextFile](#)

[stats \[Double\]](#)

[sortBy](#)

[sortByKey \[Ordered\]](#)

[stdev \[Double\]](#), [sampleStdev \[Double\]](#)

[subtract](#)

[subtractByKey \[Pair\]](#)

[sum \[Double\]](#), [sumApprox\[Double\]](#)

[take](#)

[takeOrdered](#)

[takeSample](#)

[treeAggregate](#)

[treeReduce](#)

[toDebugString](#)

[toJavaRDD](#)

[toLocalIterator](#)

[top](#)

[toString](#)

[union, ++](#)

[unpersist](#)

[values \[Pair\]](#)

[variance \[Double\]](#), [sampleVariance \[Double\]](#)

[zip](#)

[zipPartitions](#)

[zipWithIndex](#)

[zipWithUniquId](#)

## aggregateByKey [Pair]

Works like the aggregate function except the aggregation is applied to the values with the same key. Also unlike the agg

### Listing Variants

```
def aggregateByKey[U](zeroValue: U)(seqOp: (U, V) => U, combOp: (U, U) => U)(implicit arg0: ClassTag[U]): F
def aggregateByKey[U](zeroValue: U, numPartitions: Int)(seqOp: (U, V) => U, combOp: (U, U) => U)(implicit ar
def aggregateByKey[U](zeroValue: U, partitioner: Partitioner)(seqOp: (U, V) => U, combOp: (U, U) => U)(implic
```

### Example

```
val pairRDD = sc.parallelize(List( ("cat",2), ("cat", 5), ("mouse", 4),("cat", 12), ("dog",
12), ("mouse", 2)), 2)

// lets have a look at what is in the partitions
def myfunc(index: Int, iter: Iterator[(String, Int)]) : Iterator[String] = {
  iter.toList.map(x => "[partID:" + index + ", val: " + x + "]").iterator
}
pairRDD.mapPartitionsWithIndex(myfunc).collect

res2: Array[String] = Array([partID:0, val: (cat,2)], [partID:0, val: (cat,5)], [partID:0,
val: (mouse,4)], [partID:1, val: (cat,12)], [partID:1, val: (dog,12)], [partID:1, val:
(mouse,2)])

pairRDD.aggregateByKey(0)(math.max(_,_), _+_).collect
res3: Array[(String, Int)] = Array((dog,12), (cat,17), (mouse,6))

pairRDD.aggregateByKey(100)(math.max(_,_), _+_).collect
res4: Array[(String, Int)] = Array((dog,100), (cat,200), (mouse,200))
```

## cartesian

Computes the cartesian product between two RDDs (i.e. Each item of the first RDD is joined with each item of the seco

### Listing Variants

```
def cartesian[U: ClassTag](other: RDD[U]): RDD[(T, U)]
```

### Example

```
val x = sc.parallelize(List(1,2,3,4,5))
val y = sc.parallelize(List(6,7,8,9,10))
x.cartesian(y).collect
res0: Array[(Int, Int)] = Array((1,6), (1,7), (1,8), (1,9), (1,10), (2,6), (2,7),
(2,8), (2,9), (2,10), (3,6), (3,7), (3,8), (3,9), (3,10), (4,6), (5,6), (4,7), (5,7),
(4,8), (5,8), (4,9), (4,10), (5,9), (5,10))
```

## checkpoint

Will create a checkpoint when the RDD is computed next. Checkpointed RDDs are stored as a binary file within the che  
*invoked.*)

Important note: the directory "my\_directory\_name" should exist in all slaves. As an alternative you could use an HDFS

### Listing Variants

```
def checkpoint()
```

### Example

```
sc.setCheckpointDir("my_directory_name")
val a = sc.parallelize(1 to 4)
a.checkpoint
a.count
14/02/25 18:13:53 INFO SparkContext: Starting job: count at <console>:15
...
14/02/25 18:13:53 INFO MemoryStore: Block broadcast_5 stored as values to
```

```
memory (estimated size 115.7 KB, free 296.3 MB)
14/02/25 18:13:53 INFO RDDCheckpointData: Done checkpointing RDD 11
to file:/home/cloudera/Documents/spark-0.9.0-incubating-bin-
cdh4/bin/my_directory_name/65407913-fdc6-4ec1-82c9-48a1656b95d6/rdd-
11, new parent is RDD 12
res23: Long = 4
```

## coalesce, repartition

Coalesces the associated data into a given number of partitions. *repartition(numPartitions)* is simply an abbreviation for

### Listing Variants

```
def coalesce ( numPartitions : Int , shuffle : Boolean = false ) : RDD [T]
def repartition ( numPartitions : Int ) : RDD [T]
```

### Example

```
val y = sc.parallelize(1 to 10, 10)
val z = y.coalesce(2, false)
z.partitions.length
res9: Int = 2
```

## cogroup [Pair], groupWith [Pair]

A very powerful set of functions that allow grouping up to 3 key-value RDDs together using their keys.

### Listing Variants

```
def cogroup[W](other: RDD[(K, W)]): RDD[(K, (Iterable[V], Iterable[W]))]
def cogroup[W](other: RDD[(K, W)], numPartitions: Int): RDD[(K, (Iterable[V], Iterable[W]))]
def cogroup[W](other: RDD[(K, W)], partitioner: Partitioner): RDD[(K, (Iterable[V], Iterable[W]))]
def cogroup[W1, W2](other1: RDD[(K, W1)], other2: RDD[(K, W2)]): RDD[(K, (Iterable[V], Iterable[W1], Iterable[W2]))]
def cogroup[W1, W2](other1: RDD[(K, W1)], other2: RDD[(K, W2)], numPartitions: Int): RDD[(K, (Iterable[V], Iterable[W1], Iterable[W2]))]
def cogroup[W1, W2](other1: RDD[(K, W1)], other2: RDD[(K, W2)], partitioner: Partitioner): RDD[(K, (Iterable[V], Iterable[W1], Iterable[W2]))]
def groupWith[W](other: RDD[(K, W)]): RDD[(K, (Iterable[V], Iterable[W]))]
def groupWith[W1, W2](other1: RDD[(K, W1)], other2: RDD[(K, W2)]): RDD[(K, (Iterable[V], Iterable[W1], Iterable[W2]))]
```

### Examples

```
val a = sc.parallelize(List(1, 2, 1, 3), 1)
val b = a.map(_,"b")
val c = a.map(_,"c")
b.cogroup(c).collect
res7: Array[(Int, (Iterable[String], Iterable[String]))] = Array(
(2,(ArrayBuffer(b),ArrayBuffer(c))),
(3,(ArrayBuffer(b),ArrayBuffer(c))),
(1,(ArrayBuffer(b, b),ArrayBuffer(c, c)))
)

val d = a.map(_,"d")
b.cogroup(c, d).collect
res9: Array[(Int, (Iterable[String], Iterable[String], Iterable[String]))] = Array(
(2,(ArrayBuffer(b),ArrayBuffer(c),ArrayBuffer(d))),
(3,(ArrayBuffer(b),ArrayBuffer(c),ArrayBuffer(d))),
(1,(ArrayBuffer(b, b),ArrayBuffer(c, c),ArrayBuffer(d, d)))
)

val x = sc.parallelize(List((1, "apple"), (2, "banana"), (3, "orange"), (4, "kiwi")), 2)
val y = sc.parallelize(List((5, "computer"), (1, "laptop"), (1, "desktop"), (4, "iPad")), 2)
x.cogroup(y).collect
res23: Array[(Int, (Iterable[String], Iterable[String]))] = Array(
(4,(ArrayBuffer(kiwi),ArrayBuffer(iPad))),
(2,(ArrayBuffer(banana),ArrayBuffer(c))),
(3,(ArrayBuffer(orange),ArrayBuffer(c))),
(1,(ArrayBuffer(apple),ArrayBuffer(laptop, desktop))),
(5,(ArrayBuffer(),ArrayBuffer(computer)))
)
```

## collect, toArray

Converts the RDD into a Scala array and returns it. If you provide a standard map-function (i.e.  $f = T \rightarrow U$ ) it will be ap

### Listing Variants

```
def collect(): Array[T]
def collect[U: ClassTag](f: PartialFunction[T, U]): RDD[U]
def toArray(): Array[T]
```

### Example

```
val c = sc.parallelize(List("Gnu", "Cat", "Rat", "Dog", "Gnu", "Rat"), 2)
c.collect
res29: Array[String] = Array(Gnu, Cat, Rat, Dog, Gnu, Rat)
```

## collectAsMap [Pair]

Similar to *collect*, but works on key-value RDDs and converts them into Scala maps to preserve their key-value structur

### Listing Variants

```
def collectAsMap(): Map[K, V]
```

### Example

```
val a = sc.parallelize(List(1, 2, 1, 3), 1)
val b = a.zip(a)
b.collectAsMap
res1: scala.collection.Map[Int,Int] = Map(2 -> 2, 1 -> 1, 3 -> 3)
```

## combineByKey[Pair]

Very efficient implementation that combines the values of a RDD consisting of two-component tuples by applying mult

### Listing Variants

```
def combineByKey[C](createCombiner: V => C, mergeValue: (C, V) => C, mergeCombiners: (C, C) => C): RDI
def combineByKey[C](createCombiner: V => C, mergeValue: (C, V) => C, mergeCombiners: (C, C) => C, numF
def combineByKey[C](createCombiner: V => C, mergeValue: (C, V) => C, mergeCombiners: (C, C) => C, partit
```

### Example

```
val a =
sc.parallelize(List("dog", "cat", "gnu", "salmon", "rabbit", "turkey", "wolf", "bear", "bee"), 3)
val b = sc.parallelize(List(1, 1, 2, 2, 2, 1, 2, 2, 2), 3)
val c = b.zip(a)
val d = c.combineByKey(List(), (x: List[String], y: String) => y :: x, (x: List[String],
y: List[String]) => x ::: y)
d.collect
res16: Array[(Int, List[String])] = Array((1, List(cat, dog, turkey)), (2, List(gnu, rabbit,
salmon, bee, bear, wolf)))
```

## compute

Executes dependencies and computes the actual representation of the RDD. This function should not be called directly t

### Listing Variants

```
def compute(split: Partition, context: TaskContext): Iterator[T]
```

## context, sparkContext

Returns the *SparkContext* that was used to create the RDD.

### Listing Variants

```
def compute(split: Partition, context: TaskContext): Iterator[T]
```

### Example

```
val c = sc.parallelize(List("Gnu", "Cat", "Rat", "Dog"), 2)
c.context
res8: org.apache.spark.SparkContext = org.apache.spark.SparkContext@58c1c2f1
```

## count

Returns the number of items stored within a RDD.

### Listing Variants

```
def count(): Long
```

### Example

```
val c = sc.parallelize(List("Gnu", "Cat", "Rat", "Dog"), 2)
c.count
res2: Long = 4
```

## countApprox

Marked as experimental feature! Experimental features are currently not covered by this document!

### Listing Variants

```
def (timeout: Long, confidence: Double = 0.95): PartialResult[BoundedDouble]
```

## countApproxDistinct

Computes the approximate number of distinct values. For large RDDs which are spread across many nodes, this function

### Listing Variants

```
def countApproxDistinct(relativeSD: Double = 0.05): Long
```

### Example

```
val a = sc.parallelize(1 to 10000, 20)
val b = a++a++a++a++a
b.countApproxDistinct(0.1)
res14: Long = 8224

b.countApproxDistinct(0.05)
res15: Long = 9750

b.countApproxDistinct(0.01)
res16: Long = 9947

b.countApproxDistinct(0.001)
res0: Long = 10000
```

## countApproxDistinctByKey [Pair]

Similar to *countApproxDistinct*, but computes the approximate number of distinct values for each distinct key. Hence, the

methods. The parameter *relativeSD* controls the accuracy of the computation.

#### Listing Variants

```
def countApproxDistinctByKey(relativeSD: Double = 0.05): RDD[(K, Long)]
def countApproxDistinctByKey(relativeSD: Double, numPartitions: Int): RDD[(K, Long)]
def countApproxDistinctByKey(relativeSD: Double, partitioner: Partitioner): RDD[(K, Long)]
```

#### Example

```
val a = sc.parallelize(List("Gnu", "Cat", "Rat", "Dog"), 2)
val b = sc.parallelize(a.takeSample(true, 10000, 0), 20)
val c = sc.parallelize(1 to b.count().toInt, 20)
val d = b.zip(c)
d.countApproxDistinctByKey(0.1).collect
res15: Array[(String, Long)] = Array((Rat,2567), (Cat,3357), (Dog,2414), (Gnu,2494))

d.countApproxDistinctByKey(0.01).collect
res16: Array[(String, Long)] = Array((Rat,2555), (Cat,2455), (Dog,2425), (Gnu,2513))

d.countApproxDistinctByKey(0.001).collect
res0: Array[(String, Long)] = Array((Rat,2562), (Cat,2464), (Dog,2451), (Gnu,2521))
```

## countByKey [Pair]

Very similar to count, but counts the values of a RDD consisting of two-component tuples for each distinct key separate

#### Listing Variants

```
def countByKey(): Map[K, Long]
```

#### Example

```
val c = sc.parallelize(List((3, "Gnu"), (3, "Yak"), (5, "Mouse"), (3, "Dog")), 2)
c.countByKey
res3: scala.collection.Map[Int,Long] = Map(3 -> 3, 5 -> 1)
```

## countByKeyApprox [Pair]

Marked as experimental feature! Experimental features are currently not covered by this document!

#### Listing Variants

```
def countByKeyApprox(timeout: Long, confidence: Double = 0.95): PartialResult[Map[K, BoundedDouble]]
```

## countByValue

Returns a map that contains all unique values of the RDD and their respective occurrence counts. (*Warning: This operat*

#### Listing Variants

```
def countByValue(): Map[T, Long]
```

#### Example

```
val b = sc.parallelize(List(1,2,3,4,5,6,7,8,2,4,2,1,1,1,1))
b.countByValue
res27: scala.collection.Map[Int,Long] = Map(5 -> 1, 8 -> 1, 3 -> 1, 6 -> 1, 1 -> 6, 2 -> 3, 4 -> 2, 7 -> 1)
```

## countByValueApprox

Marked as experimental feature! Experimental features are currently not covered by this document!

### Listing Variants

```
def countByValueApprox(timeout: Long, confidence: Double = 0.95): PartialResult[Map[T, BoundedDouble]]
```

---

## dependencies

Returns the RDD on which this RDD depends.

### Listing Variants

```
final def dependencies: Seq[Dependency[_]]
```

### Example

```
val b = sc.parallelize(List(1,2,3,4,5,6,7,8,2,4,2,1,1,1,1))
b: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[32] at parallelize at
<console>:12
b.dependencies.length
Int = 0

b.map(a => a).dependencies.length
res40: Int = 1

b.cartesian(a).dependencies.length
res41: Int = 2

b.cartesian(a).dependencies
res42: Seq[org.apache.spark.Dependency[_]] =
List(org.apache.spark.rdd.CartesianRDD$$anon$1@576ddaaa,
org.apache.spark.rdd.CartesianRDD$$anon$2@6d2efbbd)
```

---

## distinct

Returns a new RDD that contains each unique value only once.

### Listing Variants

```
def distinct(): RDD[T]
def distinct(numPartitions: Int): RDD[T]
```

### Example

```
val c = sc.parallelize(List("Gnu", "Cat", "Rat", "Dog", "Gnu", "Rat"), 2)
c.distinct.collect
res6: Array[String] = Array(Dog, Gnu, Cat, Rat)

val a = sc.parallelize(List(1,2,3,4,5,6,7,8,9,10))
a.distinct(2).partitions.length
res16: Int = 2

a.distinct(3).partitions.length
res17: Int = 3
```

---

## first

Looks for the very first data item of the RDD and returns it.

### Listing Variants

```
def first(): T
```

### Example

```
val c = sc.parallelize(List("Gnu", "Cat", "Rat", "Dog"), 2)
c.first
res1: String = Gnu
```

## filter

Evaluates a boolean function for each data item of the RDD and puts the items for which the function returned *true* into

### Listing Variants

```
def filter(f: T => Boolean): RDD[T]
```

### Example

```
val a = sc.parallelize(1 to 10, 3)
val b = a.filter(_ % 2 == 0)
b.collect
res3: Array[Int] = Array(2, 4, 6, 8, 10)
```

When you provide a filter function, it must be able to handle all data items contained in the RDD. Scala provides so-called *isDefinedAt* for the good data (matching data) you want to apply some kind of map function. The following article is good. It tea

### Examples for mixed data without partial functions

```
val b = sc.parallelize(1 to 8)
b.filter(_ < 4).collect
res15: Array[Int] = Array(1, 2, 3)

val a = sc.parallelize(List("cat", "horse", 4.0, 3.5, 2, "dog"))
a.filter(_ < 4).collect
<console>:15: error: value < is not a member of Any
```

This fails because some components of *a* are not implicitly comparable against integers. Collect uses the *isDefinedAt* pr using the function-object.

### Examples for mixed data with partial functions

```
val a = sc.parallelize(List("cat", "horse", 4.0, 3.5, 2, "dog"))
a.collect( {case a: Int => "is integer" |
           case b: String => "is string" } ).collect
res17: Array[String] = Array(is string, is string, is integer, is string)

val myfunc: PartialFunction[Any, Any] = {
  case a: Int => "is integer" |
  case b: String => "is string" }
myfunc.isDefinedAt("")
res21: Boolean = true

myfunc.isDefinedAt(1)
res22: Boolean = true

myfunc.isDefinedAt(1.5)
res23: Boolean = false
```

Be careful! The above code works because it only checks the type itself! If you use operations on this type, you have to

```
val myfunc2: PartialFunction[Any, Any] = {case x if (x < 4) => "x"}
<console>:10: error: value < is not a member of Any

val myfunc2: PartialFunction[Int, Any] = {case x if (x < 4) => "x"}
myfunc2: PartialFunction[Int,Any] = <function1>
```

## filterByRange [Ordered]

Returns an RDD containing only the items in the key range specified. From our testing, it appears this only works if you

### Listing Variants

```
def filterByRange(lower: K, upper: K): RDD[P]
```

**Example**

```
val randRDD = sc.parallelize(List((2,"cat"), (6, "mouse"),(7, "cup"), (3, "book"), (4, "tv"), (1,
"screen"), (5, "heater")), 3)
val sortedRDD = randRDD.sortByKey()

sortedRDD.filterByRange(1, 3).collect
res66: Array[(Int, String)] = Array((1,screen), (2,cat), (3,book))
```

**filterWith (deprecated)**

This is an extended version of *filter*. It takes two function arguments. The first argument must conform to *Int -> T* and *i* is index and *U* are the data items from the RDD. Finally the function has to return either true or false (*i.e. Apply the filter*).

**Listing Variants**

```
def filterWith[A: ClassTag](constructA: Int => A)(p: (T, A) => Boolean): RDD[T]
```

**Example**

```
val a = sc.parallelize(1 to 9, 3)
val b = a.filterWith(i => i)((x,i) => x % 2 == 0 || i % 2 == 0)
b.collect
res37: Array[Int] = Array(1, 2, 3, 4, 6, 7, 8, 9)

val a = sc.parallelize(List(1,2,3,4,5,6,7,8,9,10), 5)
a.filterWith(x=> x)((a, b) => b == 0).collect
res30: Array[Int] = Array(1, 2)

a.filterWith(x=> x)((a, b) => a % (b+1) == 0).collect
res33: Array[Int] = Array(1, 2, 4, 6, 8, 10)

a.filterWith(x=> x.toString)((a, b) => b == "2").collect
res34: Array[Int] = Array(5, 6)
```

**flatMap**

Similar to *map*, but allows emitting more than one item in the map function.

**Listing Variants**

```
def flatMap[U: ClassTag](f: T => TraversableOnce[U]): RDD[U]
```

**Example**

```
val a = sc.parallelize(1 to 10, 5)
a.flatMap(1 to _).collect
res47: Array[Int] = Array(1, 1, 2, 1, 2, 3, 1, 2, 3, 4, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 6, 1, 2, 3, 4,
5, 6, 7, 1, 2, 3, 4, 5, 6, 7, 8, 1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

sc.parallelize(List(1, 2, 3), 2).flatMap(x => List(x, x, x)).collect
res85: Array[Int] = Array(1, 1, 1, 2, 2, 2, 3, 3, 3)

// The program below generates a random number of copies (up to 10) of the items in the
list.
val x = sc.parallelize(1 to 10, 3)
x.flatMap(List.fill(scala.util.Random.nextInt(10))(_)).collect

res1: Array[Int] = Array(1, 2, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 5, 5, 6, 6, 6, 6, 6, 6, 6, 7, 7,
7, 8, 8, 8, 8, 8, 8, 8, 9, 9, 9, 9, 9, 10, 10, 10, 10, 10, 10, 10, 10)
```

**flatMapValues**

Very similar to *mapValues*, but collapses the inherent structure of the values during mapping.

**Listing Variants**

```
def flatMapValues[U](f: V => TraversableOnce[U]): RDD[(K, U)]
```

**Example**

```
val a = sc.parallelize(List("dog", "tiger", "lion", "cat", "panther", "eagle"), 2)
val b = a.map(x => (x.length, x))
b.flatMapValues("x" + _ + "x").collect
res6: Array[(Int, Char)] = Array((3,x), (3,d), (3,o), (3,g), (3,x), (5,x), (5,t), (5,i), (5,g),
(5,e), (5,r), (5,x), (4,x), (4,l), (4,i), (4,o), (4,n), (4,x), (3,x), (3,e), (3,a), (3,t), (3,x), (7,x),
(7,p), (7,a), (7,n), (7,t), (7,h), (7,e), (7,r), (7,x), (5,x), (5,e), (5,a), (5,g), (5,l), (5,e), (5,x))
```

---

**flatMapWith (deprecated)**

Similar to *flatMap*, but allows accessing the partition index or a derivative of the partition index from within the flatMap

**Listing Variants**

```
def flatMapWith[A: ClassTag, U: ClassTag](constructA: Int => A, preservesPartitioning: Boolean = false)(f: (T, I
```

**Example**

```
val a = sc.parallelize(List(1,2,3,4,5,6,7,8,9), 3)
a.flatMapWith(x => x, true)((x, y) => List(y, x)).collect
res58: Array[Int] = Array(0, 1, 0, 2, 0, 3, 1, 4, 1, 5, 1, 6, 2, 7, 2, 8, 2, 9)
```

---

**fold**

Aggregates the values of each partition. The aggregation variable within each partition is initialized with *zeroValue*.

**Listing Variants**

```
def fold(zeroValue: T)(op: (T, T) => T): T
```

**Example**

```
val a = sc.parallelize(List(1,2,3), 3)
a.fold(0)(_ + _)
res59: Int = 6
```

---

**foldByKey [Pair]**

Very similar to *fold*, but performs the folding separately for each key of the RDD. This function is only available if the I

**Listing Variants**

```
def foldByKey(zeroValue: V)(func: (V, V) => V): RDD[(K, V)]
def foldByKey(zeroValue: V, numPartitions: Int)(func: (V, V) => V): RDD[(K, V)]
def foldByKey(zeroValue: V, partitioner: Partitioner)(func: (V, V) => V): RDD[(K, V)]
```

**Example**

```
val a = sc.parallelize(List("dog", "cat", "owl", "gnu", "ant"), 2)
val b = a.map(x => (x.length, x))
b.foldByKey("")( _ + _ ).collect
res84: Array[(Int, String)] = Array((3,dogcatowlgnuant)

val a = sc.parallelize(List("dog", "tiger", "lion", "cat", "panther", "eagle"), 2)
val b = a.map(x => (x.length, x))
b.foldByKey("")( _ + _ ).collect
res85: Array[(Int, String)] = Array((4,lion), (3,dogcat), (7,panther), (5,tigereagle))
```

---

## foreach

Executes an parameterless function for each data item.

### Listing Variants

```
def foreach(f: T => Unit)
```

### Example

```
val c = sc.parallelize(List("cat", "dog", "tiger", "lion", "gnu", "crocodile", "ant", "whale",  
"dolphin", "spider"), 3)  
c.foreach(x => println(x + "s are yummy"))  
lions are yummy  
gnus are yummy  
crocodiles are yummy  
ants are yummy  
whales are yummy  
dolphins are yummy  
spiders are yummy
```

---

## foreachPartition

Executes an parameterless function for each partition. Access to the data items contained in the partition is provided via

### Listing Variants

```
def foreachPartition(f: Iterator[T] => Unit)
```

### Example

```
val b = sc.parallelize(List(1, 2, 3, 4, 5, 6, 7, 8, 9), 3)  
b.foreachPartition(x => println(x.reduce(_ + _)))  
6  
15  
24
```

---

## foreachWith (Deprecated)

Executes an parameterless function for each partition. Access to the data items contained in the partition is provided via

### Listing Variants

```
def foreachWith[A: ClassTag](constructA: Int => A)(f: (T, A) => Unit)
```

### Example

```
val a = sc.parallelize(1 to 9, 3)  
a.foreachWith(i => i)((x,i) => if (x % 2 == 1 && i % 2 == 0) println(x) )  
1  
3  
7  
9
```

---

## fullOuterJoin [Pair]

Performs the full outer join between two paired RDDs.

### Listing Variants

```
def fullOuterJoin[W](other: RDD[(K, W)], numPartitions: Int): RDD[(K, (Option[V], Option[W]))]  
def fullOuterJoin[W](other: RDD[(K, W)]): RDD[(K, (Option[V], Option[W]))]
```

```
def fullOuterJoin[W](other: RDD[(K, W)], partitioner: Partitioner): RDD[(K, (Option[V], Option[W]))]
```

**Example**

```
val pairRDD1 = sc.parallelize(List( ("cat",2), ("cat", 5), ("book", 4),("cat", 12)))
val pairRDD2 = sc.parallelize(List( ("cat",2), ("cup", 5), ("mouse", 4),("cat", 12)))
pairRDD1.fullOuterJoin(pairRDD2).collect

res5: Array[(String, (Option[Int], Option[Int]))] = Array((book,(Some(4),None)), (mouse,
(None,Some(4))), (cup,(None,Some(5))), (cat,(Some(2),Some(2))), (cat,(Some(2),Some(12))),
(cat,(Some(5),Some(2))), (cat,(Some(5),Some(12))), (cat,(Some(12),Some(2))), (cat,
(Some(12),Some(12))))
```

**generator, setGenerator**

Allows setting a string that is attached to the end of the RDD's name when printing the dependency graph.

**Listing Variants**

```
@transient var generator
def setGenerator(_generator: String)
```

**getCheckpointFile**

Returns the path to the checkpoint file or null if RDD has not yet been checkpointed.

**Listing Variants**

```
def getCheckpointFile: Option[String]
```

**Example**

```
sc.setCheckpointDir("/home/cloudera/Documents")
val a = sc.parallelize(1 to 500, 5)
val b = a++a++a++a++a
b.getCheckpointFile
res49: Option[String] = None

b.checkpoint
b.getCheckpointFile
res54: Option[String] = None

b.collect
b.getCheckpointFile
res57: Option[String] = Some(file:/home/cloudera/Documents/cb978ffb-a346-4820-
b3ba-d56580787b20/rdd-40)
```

**preferredLocations**

Returns the hosts which are preferred by this RDD. The actual preference of a specific host depends on various assumpt

**Listing Variants**

```
final def preferredLocations(split: Partition): Seq[String]
```

**getStorageLevel**

Retrieves the currently set storage level of the RDD. This can only be used to assign a new storage level if the RDD doe

**Listing Variants**

```
def getStorageLevel
```

**Example**

```
val a = sc.parallelize(1 to 100000, 2)
a.persist(org.apache.spark.storage.StorageLevel.DISK_ONLY)
a.getStorageLevel.description
String = Disk Serialized 1x Replicated

a.cache
java.lang.UnsupportedOperationException: Cannot change storage level of an RDD after
it was already assigned a level
```

**glom**

Assembles an array that contains all elements of the partition and embeds it in an RDD. Each returned array contains the

**Listing Variants**

```
def glom(): RDD[Array[T]]
```

**Example**

```
val a = sc.parallelize(1 to 100, 3)
a.glom.collect
res8: Array[Array[Int]] = Array(Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33), Array(34, 35, 36, 37, 38,
39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62,
63, 64, 65, 66), Array(67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84,
85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100))
```

**groupBy****Listing Variants**

```
def groupBy[K: ClassTag](f: T => K): RDD[(K, Iterable[T])]
def groupBy[K: ClassTag](f: T => K, numPartitions: Int): RDD[(K, Iterable[T])]
def groupBy[K: ClassTag](f: T => K, p: Partitioner): RDD[(K, Iterable[T])]
```

**Example**

```
val a = sc.parallelize(1 to 9, 3)
a.groupBy(x => { if (x % 2 == 0) "even" else "odd" }).collect
res42: Array[(String, Seq[Int])] = Array((even, ArrayBuffer(2, 4, 6, 8)),
(odds, ArrayBuffer(1, 3, 5, 7, 9)))

val a = sc.parallelize(1 to 9, 3)
def myfunc(a: Int) : Int =
{
  a % 2
}
a.groupBy(myfunc).collect
res3: Array[(Int, Seq[Int])] = Array((0, ArrayBuffer(2, 4, 6, 8)), (1, ArrayBuffer(1, 3, 5, 7, 9)))

val a = sc.parallelize(1 to 9, 3)
def myfunc(a: Int) : Int =
{
  a % 2
}
a.groupBy(x => myfunc(x), 3).collect
a.groupBy(myfunc(_), 1).collect
res7: Array[(Int, Seq[Int])] = Array((0, ArrayBuffer(2, 4, 6, 8)), (1, ArrayBuffer(1, 3, 5, 7, 9)))

import org.apache.spark.Partitioner
class MyPartitioner extends Partitioner {
  def numPartitions: Int = 2
  def getPartition(key: Any): Int =
  {
    key match
```

```

{
  case null => 0
  case key: Int => key % numPartitions
  case _ => key.hashCode % numPartitions
}
}
override def equals(other: Any): Boolean =
{
  other match
  {
    case h: MyPartitioner => true
    case _ => false
  }
}
}
val a = sc.parallelize(1 to 9, 3)
val p = new MyPartitioner()
val b = a.groupBy((x: Int) => { x }, p)
val c = b.mapWith(i => i)((a, b) => (b, a))
c.collect
res42: Array[(Int, (Int, Seq[Int]))] = Array((0,(4,ArrayBuffer(4))), (0,(2,ArrayBuffer(2))),
(0,(6,ArrayBuffer(6))), (0,(8,ArrayBuffer(8))), (1,(9,ArrayBuffer(9))), (1,
(3,ArrayBuffer(3))), (1,(1,ArrayBuffer(1))), (1,(7,ArrayBuffer(7))), (1,
(5,ArrayBuffer(5))))

```

## groupByKey [Pair]

Very similar to *groupBy*, but instead of supplying a function, the key-component of each pair will automatically be preserved.

### Listing Variants

```

def groupByKey(): RDD[(K, Iterable[V])]
def groupByKey(numPartitions: Int): RDD[(K, Iterable[V])]
def groupByKey(partitioner: Partitioner): RDD[(K, Iterable[V])]

```

### Example

```

val a = sc.parallelize(List("dog", "tiger", "lion", "cat", "spider", "eagle"), 2)
val b = a.keyBy(_.length)
b.groupByKey.collect
res11: Array[(Int, Seq[String])] = Array((4,ArrayBuffer(lion)), (6,ArrayBuffer(spider)),
(3,ArrayBuffer(dog, cat)), (5,ArrayBuffer(tiger, eagle)))

```

## histogram [Double]

These functions take an RDD of doubles and create a histogram with either even spacing (the number of buckets equals slightly different, the first function will return a tuple consisting of two arrays. The first array contains the computed bucket boundaries and the second array contains the histogram as an array of integers.

### Listing Variants

```

def histogram(bucketCount: Int): Pair[Array[Double], Array[Long]]
def histogram(buckets: Array[Double], evenBuckets: Boolean = false): Array[Long]

```

### Example with even spacing

```

val a = sc.parallelize(List(1.1, 1.2, 1.3, 2.0, 2.1, 7.4, 7.5, 7.6, 8.8, 9.0), 3)
a.histogram(5)
res11: (Array[Double], Array[Long]) = (Array(1.1, 2.68, 4.26, 5.84, 7.42, 9.0), Array(5,
0, 0, 1, 4))

val a = sc.parallelize(List(9.1, 1.0, 1.2, 2.1, 1.3, 5.0, 2.0, 2.1, 7.4, 7.5, 7.6, 8.8, 10.0, 8.9,
5.5), 3)
a.histogram(6)
res18: (Array[Double], Array[Long]) = (Array(1.0, 2.5, 4.0, 5.5, 7.0, 8.5, 10.0), Array(6,
0, 1, 1, 3, 4))

```

### Example with custom spacing

```

val a = sc.parallelize(List(1.1, 1.2, 1.3, 2.0, 2.1, 7.4, 7.5, 7.6, 8.8, 9.0), 3)
a.histogram(Array(0.0, 3.0, 8.0))

```

```
res14: Array[Long] = Array(5, 3)

val a = sc.parallelize(List(9.1, 1.0, 1.2, 2.1, 1.3, 5.0, 2.0, 2.1, 7.4, 7.5, 7.6, 8.8, 10.0, 8.9, 5.5), 3)
a.histogram(Array(0.0, 5.0, 10.0))
res1: Array[Long] = Array(6, 9)

a.histogram(Array(0.0, 5.0, 10.0, 15.0))
res1: Array[Long] = Array(6, 8, 1)
```

## id

Retrieves the ID which has been assigned to the RDD by its device context.

### Listing Variants

```
val id: Int
```

### Example

```
val y = sc.parallelize(1 to 10, 10)
y.id
res16: Int = 19
```

## intersection

Returns the elements in the two RDDs which are the same.

### Listing Variants

```
def intersection(other: RDD[T], numPartitions: Int): RDD[T]
def intersection(other: RDD[T], partitioner: Partitioner)(implicit ord: Ordering[T] = null): RDD[T]
def intersection(other: RDD[T]): RDD[T]
```

### Example

```
val x = sc.parallelize(1 to 20)
val y = sc.parallelize(10 to 30)
val z = x.intersection(y)

z.collect
res74: Array[Int] = Array(16, 12, 20, 13, 17, 14, 18, 10, 19, 15, 11)
```

## isCheckpointed

Indicates whether the RDD has been checkpointed. The flag will only raise once the checkpoint has really been created.

### Listing Variants

```
def isCheckpointed: Boolean
```

### Example

```
sc.setCheckpointDir("/home/cloudera/Documents")
c.isCheckpointed
res6: Boolean = false

c.checkpoint
c.isCheckpointed
res8: Boolean = false

c.collect
c.isCheckpointed
res9: Boolean = true
```

## iterator

Returns a compatible iterator object for a partition of this RDD. This function should never be called directly.

### Listing Variants

```
final def iterator(split: Partition, context: TaskContext): Iterator[T]
```

---

## join [Pair]

Performs an inner join using two key-value RDDs. Please note that the keys must be generally comparable to make this

### Listing Variants

```
def join[W](other: RDD[(K, W)]): RDD[(K, (V, W))]  
def join[W](other: RDD[(K, W)], numPartitions: Int): RDD[(K, (V, W))]  
def join[W](other: RDD[(K, W)], partitioner: Partitioner): RDD[(K, (V, W))]
```

### Example

```
val a = sc.parallelize(List("dog", "salmon", "salmon", "rat", "elephant"), 3)  
val b = a.keyBy(_.length)  
val c = sc.parallelize(List("dog", "cat", "gnu", "salmon", "rabbit", "turkey", "wolf", "bear", "bee"), 3)  
val d = c.keyBy(_.length)  
b.join(d).collect  
  
res0: Array[(Int, (String, String))] = Array((6,(salmon,salmon)), (6,(salmon,rabbit)), (6,  
(salmon,turkey)), (6,(salmon,salmon)), (6,(salmon,rabbit)), (6,(salmon,turkey)), (3,(dog,dog)), (3,  
(dog,cat)), (3,(dog,gnu)), (3,(dog,bee)), (3,(rat,dog)), (3,(rat,cat)), (3,(rat,gnu)), (3,(rat,bee)))
```

---

## keyBy

Constructs two-component tuples (key-value pairs) by applying a function on each data item. The result of the function

### Listing Variants

```
def keyBy[K](f: T => K): RDD[(K, T)]
```

### Example

```
val a = sc.parallelize(List("dog", "salmon", "salmon", "rat", "elephant"), 3)  
val b = a.keyBy(_.length)  
b.collect  
res26: Array[(Int, String)] = Array((3,dog), (6,salmon), (6,salmon), (3,rat), (8,elephant))
```

---

## keys [Pair]

Extracts the keys from all contained tuples and returns them in a new RDD.

### Listing Variants

```
def keys: RDD[K]
```

### Example

```
val a = sc.parallelize(List("dog", "tiger", "lion", "cat", "panther", "eagle"), 2)  
val b = a.map(x => (x.length, x))  
b.keys.collect  
res2: Array[Int] = Array(3, 5, 4, 3, 7, 5)
```

---

## leftOuterJoin [Pair]

Performs an left outer join using two key-value RDDs. Please note that the keys must be generally comparable to make

### Listing Variants

```
def leftOuterJoin[W](other: RDD[(K, W)]): RDD[(K, (V, Option[W]))]
def leftOuterJoin[W](other: RDD[(K, W)], numPartitions: Int): RDD[(K, (V, Option[W]))]
def leftOuterJoin[W](other: RDD[(K, W)], partitioner: Partitioner): RDD[(K, (V, Option[W]))]
```

### Example

```
val a = sc.parallelize(List("dog", "salmon", "salmon", "rat", "elephant"), 3)
val b = a.keyBy(_.length)
val c =
  sc.parallelize(List("dog", "cat", "gnu", "salmon", "rabbit", "turkey", "wolf", "bear", "bee"), 3)
val d = c.keyBy(_.length)
b.leftOuterJoin(d).collect

res1: Array[(Int, (String, Option[String]))] = Array((6,(salmon,Some(salmon))), (6,
(salmon,Some(rabbit))), (6,(salmon,Some(turkey))), (6,(salmon,Some(salmon))), (6,
(salmon,Some(rabbit))), (6,(salmon,Some(turkey))), (3,(dog,Some(dog))), (3,
(dog,Some(cat))), (3,(dog,Some(gnu))), (3,(dog,Some(bee))), (3,(rat,Some(dog))), (3,
(rat,Some(cat))), (3,(rat,Some(gnu))), (3,(rat,Some(bee))), (8,(elephant,None)))
```

## lookup

Scans the RDD for all keys that match the provided value and returns their values as a Scala sequence.

### Listing Variants

```
def lookup(key: K): Seq[V]
```

### Example

```
val a = sc.parallelize(List("dog", "tiger", "lion", "cat", "panther", "eagle"), 2)
val b = a.map(x => (x.length, x))
b.lookup(5)
res0: Seq[String] = WrappedArray(tiger, eagle)
```

## map

Applies a transformation function on each item of the RDD and returns the result as a new RDD.

### Listing Variants

```
def map[U: ClassTag](f: T => U): RDD[U]
```

### Example

```
val a = sc.parallelize(List("dog", "salmon", "salmon", "rat", "elephant"), 3)
val b = a.map(_.length)
val c = a.zip(b)
c.collect
res0: Array[(String, Int)] = Array((dog,3), (salmon,6), (salmon,6), (rat,3), (elephant,8))
```

## mapPartitions

This is a specialized map that is called only once for each partition. The entire content of the respective partitions is available and the iterators are automatically converted into a new RDD. Please note, that the tuples (3,4) and (6,7) are missing from the f

### Listing Variants

```
def mapPartitions[U: ClassTag](f: Iterator[T] => Iterator[U], preservesPartitioning: Boolean = false): RDD[U]
```

**Example 1**

```
val a = sc.parallelize(1 to 9, 3)
def myfunc[T](iter: Iterator[T]) : Iterator[(T, T)] = {
  var res = List[(T, T)]()
  var pre = iter.next
  while (iter.hasNext)
  {
    val cur = iter.next;
    res ::= (pre, cur)
    pre = cur;
  }
  res.iterator
}
a.mapPartitions(myfunc).collect
res0: Array[(Int, Int)] = Array((2,3), (1,2), (5,6), (4,5), (8,9), (7,8))
```

**Example 2**

```
val x = sc.parallelize(List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10), 3)
def myfunc(iter: Iterator[Int]) : Iterator[Int] = {
  var res = List[Int]()
  while (iter.hasNext) {
    val cur = iter.next;
    res = res ::: List.fill(scala.util.Random.nextInt(10))(cur)
  }
  res.iterator
}
x.mapPartitions(myfunc).collect
// some of the number are not outputted at all. This is because the random number
// generated for it is zero.
res8: Array[Int] = Array(1, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 5, 7, 7, 7, 9, 9, 10)
```

The above program can also be written using **flatMap** as follows.

**Example 2 using flatmap**

```
val x = sc.parallelize(1 to 10, 3)
x.flatMap(List.fill(scala.util.Random.nextInt(10))(_)).collect

res1: Array[Int] = Array(1, 2, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 5, 5, 6, 6, 6, 6, 6, 6, 6, 7, 7, 7, 8, 8, 8, 8, 8, 8, 8, 8, 9, 9, 9, 9, 9, 10, 10, 10, 10, 10, 10, 10, 10)
```

**mapPartitionsWithContext** (deprecated and developer API)

Similar to *mapPartitions*, but allows accessing information about the processing state within the mapper.

**Listing Variants**

```
def mapPartitionsWithContext[U: ClassTag](f: (TaskContext, Iterator[T]) => Iterator[U], preservesPartitioning: Boolean)
```

**Example**

```
val a = sc.parallelize(1 to 9, 3)
import org.apache.spark.TaskContext
def myfunc(tc: TaskContext, iter: Iterator[Int]) : Iterator[Int] = {
  tc.addOnCompleteCallback(() => println(
    "Partition: " + tc.partitionId +
    ", AttemptID: " + tc.attemptId ))

  iter.toList.filter(_ % 2 == 0).iterator
}
a.mapPartitionsWithContext(myfunc).collect

14/04/01 23:05:48 INFO SparkContext: Starting job: collect at <console>:20
...
14/04/01 23:05:48 INFO Executor: Running task ID 0
Partition: 0, AttemptID: 0, Interrupted: false
...
14/04/01 23:05:48 INFO Executor: Running task ID 1
14/04/01 23:05:48 INFO TaskSetManager: Finished TID 0 in 470 ms on localhost
(progress: 0/3)
...
14/04/01 23:05:48 INFO Executor: Running task ID 2
14/04/01 23:05:48 INFO TaskSetManager: Finished TID 1 in 23 ms on localhost
(progress: 1/3)
```

```
14/04/01 23:05:48 INFO DAGScheduler: Completed ResultTask(0, 1)
?
res0: Array[Int] = Array(2, 6, 4, 8)
```

## mapPartitionsWithIndex

Similar to *mapPartitions*, but takes two parameters. The first parameter is the index of the partition and the second is an encodes.

### Listing Variants

```
def mapPartitionsWithIndex[U: ClassTag](f: (Int, Iterator[T]) => Iterator[U], preservesPartitioning: Boolean = false)
```

### Example

```
val x = sc.parallelize(List(1,2,3,4,5,6,7,8,9,10), 3)
def myfunc(index: Int, iter: Iterator[Int]) : Iterator[String] = {
  iter.toList.map(x => index + "," + x).iterator
}
x.mapPartitionsWithIndex(myfunc).collect()
res10: Array[String] = Array(0,1, 0,2, 0,3, 1,4, 1,5, 1,6, 2,7, 2,8, 2,9, 2,10)
```

## mapPartitionsWithSplit

This method has been marked as deprecated in the API. So, you should not use this method anymore. Deprecated metho

### Listing Variants

```
def mapPartitionsWithSplit[U: ClassTag](f: (Int, Iterator[T]) => Iterator[U], preservesPartitioning: Boolean = false)
```

## mapValues [Pair]

Takes the values of a RDD that consists of two-component tuples, and applies the provided function to transform each v

### Listing Variants

```
def mapValues[U](f: V => U): RDD[(K, U)]
```

### Example

```
val a = sc.parallelize(List("dog", "tiger", "lion", "cat", "panther", "eagle"), 2)
val b = a.map(x => (x.length, x))
b.mapValues("x" + _ + "x").collect()
res5: Array[(Int, String)] = Array((3,xdogx), (5,xtigerx), (4,xlionx), (3,xcatx),
(7,xpantherx), (5,xeaglex))
```

## mapWith (deprecated)

This is an extended version of *map*. It takes two function arguments. The first argument must conform to *Int -> T* and is initialization code once per partition. Like create a Random number generator object. The second function must conform

### Listing Variants

```
def mapWith[A: ClassTag, U: ClassTag](constructA: Int => A, preservesPartitioning: Boolean = false)(f: (T, A) => U)
```

### Example

```
// generates 9 random numbers less than 1000.
```

```
val x = sc.parallelize(1 to 9, 3)
x.mapWith(a => new scala.util.Random)((x, r) => r.nextInt(1000)).collect
res0: Array[Int] = Array(940, 51, 779, 742, 757, 982, 35, 800, 15)

val a = sc.parallelize(1 to 9, 3)
val b = a.mapWith("Index:" + _)((a, b) => ("Value:" + a, b))
b.collect
res0: Array[(String, String)] = Array((Value:1,Index:0), (Value:2,Index:0),
(Value:3,Index:0), (Value:4,Index:1), (Value:5,Index:1), (Value:6,Index:1),
(Value:7,Index:2), (Value:8,Index:2), (Value:9,Index:2))
```

## max

Returns the largest element in the RDD

### Listing Variants

```
def max()(implicit ord: Ordering[T]): T
```

### Example

```
val y = sc.parallelize(10 to 30)
y.max
res75: Int = 30

val a = sc.parallelize(List((10, "dog"), (3, "tiger"), (9, "lion"), (18, "cat")))
a.max
res6: (Int, String) = (18,cat)
```

## mean [Double], meanApprox [Double]

Calls *stats* and extracts the mean component. The approximate version of the function can finish somewhat faster in son

### Listing Variants

```
def mean(): Double
def meanApprox(timeout: Long, confidence: Double = 0.95): PartialResult[BoundedDouble]
```

### Example

```
val a = sc.parallelize(List(9.1, 1.0, 1.2, 2.1, 1.3, 5.0, 2.0, 2.1, 7.4, 7.5, 7.6, 8.8, 10.0, 8.9,
5.5), 3)
a.mean
res0: Double = 5.3
```

## min

Returns the smallest element in the RDD

### Listing Variants

```
def min()(implicit ord: Ordering[T]): T
```

### Example

```
val y = sc.parallelize(10 to 30)
y.min
res75: Int = 10

val a = sc.parallelize(List((10, "dog"), (3, "tiger"), (9, "lion"), (8, "cat")))
a.min
res4: (Int, String) = (3,tiger)
```

## name, setName

Allows a RDD to be tagged with a custom name.

### Listing Variants

```
@transient var name: String
def setName(_name: String)
```

### Example

```
val y = sc.parallelize(1 to 10, 10)
y.name
res13: String = null
y.setName("Fancy RDD Name")
y.name
res15: String = Fancy RDD Name
```

---

## partitionBy [Pair]

Repartitions as key-value RDD using its keys. The partitioner implementation can be supplied as the first argument.

### Listing Variants

```
def partitionBy(partitioner: Partitioner): RDD[(K, V)]
```

---

## partitioner

Specifies a function pointer to the default partitioner that will be used for *groupBy*, *subtract*, *reduceByKey* (from *Paired*

### Listing Variants

```
@transient val partitioner: Option[Partitioner]
```

---

## partitions

Returns an array of the partition objects associated with this RDD.

### Listing Variants

```
final def partitions: Array[Partition]
```

### Example

```
val b = sc.parallelize(List("Gnu", "Cat", "Rat", "Dog", "Gnu", "Rat"), 2)
b.partitions
res48: Array[org.apache.spark.Partition] =
Array(org.apache.spark.rdd.ParallelCollectionPartition@18aa,
org.apache.spark.rdd.ParallelCollectionPartition@18ab)
```

---

## persist, cache

These functions can be used to adjust the storage level of a RDD. When freeing up memory, Spark will use the storage level *persist(StorageLevel.MEMORY\_ONLY)*. (Warning: Once the storage level has been changed, it cannot be changed again.)

### Listing Variants

```
def cache(): RDD[T]
def persist(): RDD[T]
def persist(newLevel: StorageLevel): RDD[T]
```

**Example**

```
val c = sc.parallelize(List("Gnu", "Cat", "Rat", "Dog", "Gnu", "Rat"), 2)
c.getStorageLevel
res0: org.apache.spark.storage.StorageLevel = StorageLevel(false, false, false, false, 1)
c.cache
c.getStorageLevel
res2: org.apache.spark.storage.StorageLevel = StorageLevel(false, true, false, true, 1)
```

**pipe**

Takes the RDD data of each partition and sends it via stdin to a shell-command. The resulting output of the command is

**Listing Variants**

```
def pipe(command: String): RDD[String]
def pipe(command: String, env: Map[String, String]): RDD[String]
def pipe(command: Seq[String], env: Map[String, String] = Map(), printPipeContext: (String => Unit) => Unit =
```

**Example**

```
val a = sc.parallelize(1 to 9, 3)
a.pipe("head -n 1").collect
res2: Array[String] = Array(1, 4, 7)
```

**randomSplit**

Randomly splits an RDD into multiple smaller RDDs according to a weights Array which specifies the percentage of the weights Array. The second example below shows the number of items in each smaller RDD does not exactly match

**Listing Variants**

```
def randomSplit(weights: Array[Double], seed: Long = Utils.random.nextLong): Array[RDD[T]]
```

**Example**

```
val y = sc.parallelize(1 to 10)
val splits = y.randomSplit(Array(0.6, 0.4), seed = 11L)
val training = splits(0)
val test = splits(1)
training.collect
res:85 Array[Int] = Array(1, 4, 5, 6, 8, 10)
test.collect
res86: Array[Int] = Array(2, 3, 7, 9)

val y = sc.parallelize(1 to 10)
val splits = y.randomSplit(Array(0.1, 0.3, 0.6))

val rdd1 = splits(0)
val rdd2 = splits(1)
val rdd3 = splits(2)

rdd1.collect
res87: Array[Int] = Array(4, 10)
rdd2.collect
res88: Array[Int] = Array(1, 3, 5, 8)
rdd3.collect
res91: Array[Int] = Array(2, 6, 7, 9)
```

**reduce**

This function provides the well-known *reduce* functionality in Spark. Please note that any function *f* you provide, should

**Listing Variants**

```
def reduce(f: (T, T) => T): T
```

**Example**

```
val a = sc.parallelize(1 to 100, 3)
a.reduce(_ + _)
res41: Int = 5050
```

**reduceByKey [Pair], reduceByKeyLocally [Pair], reduceByKeyToDriver [Pair]**

This function provides the well-known *reduce* functionality in Spark. Please note that any function *f* you provide, should

**Listing Variants**

```
def reduceByKey(func: (V, V) => V): RDD[(K, V)]
def reduceByKey(func: (V, V) => V, numPartitions: Int): RDD[(K, V)]
def reduceByKey(partitioner: Partitioner, func: (V, V) => V): RDD[(K, V)]
def reduceByKeyLocally(func: (V, V) => V): Map[K, V]
def reduceByKeyToDriver(func: (V, V) => V): Map[K, V]
```

**Example**

```
val a = sc.parallelize(List("dog", "cat", "owl", "gnu", "ant"), 2)
val b = a.map(x => (x.length, x))
b.reduceByKey(_ + _).collect
res86: Array[(Int, String)] = Array((3,dogcatowlgnuant))

val a = sc.parallelize(List("dog", "tiger", "lion", "cat", "panther", "eagle"), 2)
val b = a.map(x => (x.length, x))
b.reduceByKey(_ + _).collect
res87: Array[(Int, String)] = Array((4,lion), (3,dogcat), (7,panther), (5,tigereagle))
```

**repartition**

This function changes the number of partitions to the number specified by the numPartitions parameter

**Listing Variants**

```
def repartition(numPartitions: Int)(implicit ord: Ordering[T] = null): RDD[T]
```

**Example**

```
val rdd = sc.parallelize(List(1, 2, 10, 4, 5, 2, 1, 1, 1), 3)
rdd.partitions.length
res2: Int = 3
val rdd2 = rdd.repartition(5)
rdd2.partitions.length
res6: Int = 5
```

**repartitionAndSortWithinPartitions [Ordered]**

Repartition the RDD according to the given partitioner and, within each resulting partition, sort records by their keys.

**Listing Variants**

```
def repartitionAndSortWithinPartitions(partitioner: Partitioner): RDD[(K, V)]
```

**Example**

```
// first we will do range partitioning which is not sorted
val randRDD = sc.parallelize(List( (2,"cat"), (6, "mouse"),(7, "cup"), (3, "book"), (4, "tv"), (1, "screen"),
(5, "heater")), 3)
val rPartitioner = new org.apache.spark.RangePartitioner(3, randRDD)
val partitioned = randRDD.partitionBy(rPartitioner)
def myfunc(index: Int, iter: Iterator[(Int, String)]) : Iterator[String] = {
  iter.toList.map(x => "[partID:" + index + ", val: " + x + "]").iterator
}
partitioned.mapPartitionsWithIndex(myfunc).collect

res0: Array[String] = Array([partID:0, val: (2,cat)], [partID:0, val: (3,book)], [partID:0, val: (1,screen)],
[partID:1, val: (4,tv)], [partID:1, val: (5,heater)], [partID:2, val: (6,mouse)], [partID:2, val: (7,cup)])

// now lets repartition but this time have it sorted
val partitioned = randRDD.repartitionAndSortWithinPartitions(rPartitioner)
def myfunc(index: Int, iter: Iterator[(Int, String)]) : Iterator[String] = {
  iter.toList.map(x => "[partID:" + index + ", val: " + x + "]").iterator
}
partitioned.mapPartitionsWithIndex(myfunc).collect

res1: Array[String] = Array([partID:0, val: (1,screen)], [partID:0, val: (2,cat)], [partID:0, val: (3,book)],
[partID:1, val: (4,tv)], [partID:1, val: (5,heater)], [partID:2, val: (6,mouse)], [partID:2, val: (7,cup)])
```

## rightOuterJoin [Pair]

Performs an right outer join using two key-value RDDs. Please note that the keys must be generally comparable to make

### Listing Variants

```
def rightOuterJoin[W](other: RDD[(K, W)]): RDD[(K, (Option[V], W))]
def rightOuterJoin[W](other: RDD[(K, W)], numPartitions: Int): RDD[(K, (Option[V], W))]
def rightOuterJoin[W](other: RDD[(K, W)], partitioner: Partitioner): RDD[(K, (Option[V], W))]
```

### Example

```
val a = sc.parallelize(List("dog", "salmon", "salmon", "rat", "elephant"), 3)
val b = a.keyBy(_.length)
val c =
sc.parallelize(List("dog","cat","gnu","salmon","rabbit","turkey","wolf","bear","bee"), 3)
val d = c.keyBy(_.length)
b.rightOuterJoin(d).collect

res2: Array[(Int, (Option[String], String))] = Array((6,(Some(salmon),salmon)), (6,
(Some(salmon),rabbit)), (6,(Some(salmon),turkey)), (6,(Some(salmon),salmon)), (6,
(Some(salmon),rabbit)), (6,(Some(salmon),turkey)), (3,(Some(dog),dog)), (3,
(Some(dog),cat)), (3,(Some(dog),gnu)), (3,(Some(dog),bee)), (3,(Some(rat),dog)), (3,
(Some(rat),cat)), (3,(Some(rat),gnu)), (3,(Some(rat),bee)), (4,(None,wolf)), (4,
(None,bear)))
```

## sample

Randomly selects a fraction of the items of a RDD and returns them in a new RDD.

### Listing Variants

```
def sample(withReplacement: Boolean, fraction: Double, seed: Int): RDD[T]
```

### Example

```
val a = sc.parallelize(1 to 10000, 3)
a.sample(false, 0.1, 0).count
res24: Long = 960

a.sample(true, 0.3, 0).count
res25: Long = 2888
```

```
a.sample(true, 0.3, 13).count
res26: Long = 2985
```

## sampleByKey [Pair]

Randomly samples the key value pair RDD according to the fraction of each key you want to appear in the final RDD.

### Listing Variants

```
def sampleByKey(withReplacement: Boolean, fractions: Map[K, Double], seed: Long = Utils.random.nextLong):
```

### Example

```
val randRDD = sc.parallelize(List((7,"cat"), (6, "mouse"),(7, "cup"), (6, "book"), (7, "tv"), (6,
"screen"), (7, "heater")))
val sampleMap = List((7, 0.4), (6, 0.6)).toMap
randRDD.sampleByKey(false, sampleMap,42).collect
res6: Array[(Int, String)] = Array((7,cat), (6,mouse), (6,book), (6,screen), (7,heater))
```

## sampleByKeyExact [Pair, experimental]

This is labelled as experimental and so we do not document it.

### Listing Variants

```
def sampleByKeyExact(withReplacement: Boolean, fractions: Map[K, Double], seed: Long = Utils.random.nextI
```

## saveAsHadoopFile [Pair], saveAsHadoopDataset [Pair], saveAsNewAPIHadoop

Saves the RDD in a Hadoop compatible format using any Hadoop outputFormat class the user specifies.

### Listing Variants

```
def saveAsHadoopDataset(conf: JobConf)
def saveAsHadoopFile[F <: OutputFormat[K, V]](path: String)(implicit fm: ClassTag[F])
def saveAsHadoopFile[F <: OutputFormat[K, V]](path: String, codec: Class[_ <: CompressionCodec]) (implicit f
def saveAsHadoopFile(path: String, keyClass: Class[_], valueClass: Class[_], outputFormatClass: Class[_ <: Out
def saveAsHadoopFile(path: String, keyClass: Class[_], valueClass: Class[_], outputFormatClass: Class[_ <: Out
def saveAsNewAPIHadoopFile[F <: NewOutputFormat[K, V]](path: String)(implicit fm: ClassTag[F])
def saveAsNewAPIHadoopFile(path: String, keyClass: Class[_], valueClass: Class[_], outputFormatClass: Class[
```

## saveAsObjectFile

Saves the RDD in binary format.

### Listing Variants

```
def saveAsObjectFile(path: String)
```

### Example

```
val x = sc.parallelize(1 to 100, 3)
x.saveAsObjectFile("objFile")
val y = sc.objectFile[Int]("objFile")
y.collect
res52: Array[Int] = Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40,
41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64,
```

65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100)
--

## saveAsSequenceFile [SeqFile]

Saves the RDD as a Hadoop sequence file.

### Listing Variants

```
def saveAsSequenceFile(path: String, codec: Option[Class[_ <: CompressionCodec]] = None)
```

### Example

```
val v = sc.parallelize(Array(("owl",3), ("gnu",4), ("dog",1), ("cat",2), ("ant",5)), 2)
v.saveAsSequenceFile("hd_seq_file")
14/04/19 05:45:43 INFO FileOutputCommitter: Saved output of task
'attempt_201404190545_0000_m_000001_191' to file:/home/cloudera/hd_seq_file

[cloudera@localhost ~]$ ll ~/hd_seq_file
total 8
-rwxr-xr-x 1 cloudera cloudera 117 Apr 19 05:45 part-00000
-rwxr-xr-x 1 cloudera cloudera 133 Apr 19 05:45 part-00001
-rwxr-xr-x 1 cloudera cloudera  0 Apr 19 05:45 _SUCCESS
```

## saveAsTextFile

Saves the RDD as text files. One line at a time.

### Listing Variants

```
def saveAsTextFile(path: String)
def saveAsTextFile(path: String, codec: Class[_ <: CompressionCodec])
```

### Example without compression

```
val a = sc.parallelize(1 to 10000, 3)
a.saveAsTextFile("mydata_a")
14/04/03 21:11:36 INFO FileOutputCommitter: Saved output of task
'attempt_201404032111_0000_m_000002_71' to file:/home/cloudera/Documents/spark-
0.9.0-incubating-bin-cdh4/bin/mydata_a

[cloudera@localhost ~]$ head -n 5 ~/Documents/spark-0.9.0-incubating-bin-
cdh4/bin/mydata_a/part-00000
1
2
3
4
5

// Produces 3 output files since we have created the a RDD with 3 partitions
[cloudera@localhost ~]$ ll ~/Documents/spark-0.9.0-incubating-bin-cdh4/bin/mydata_a/
-rwxr-xr-x 1 cloudera cloudera 15558 Apr  3 21:11 part-00000
-rwxr-xr-x 1 cloudera cloudera 16665 Apr  3 21:11 part-00001
-rwxr-xr-x 1 cloudera cloudera 16671 Apr  3 21:11 part-00002
```

### Example with compression

```
import org.apache.hadoop.io.compress.GzipCodec
a.saveAsTextFile("mydata_b", classOf[GzipCodec])

[cloudera@localhost ~]$ ll ~/Documents/spark-0.9.0-incubating-bin-cdh4/bin/mydata_b/
total 24
-rwxr-xr-x 1 cloudera cloudera 7276 Apr  3 21:29 part-00000.gz
-rwxr-xr-x 1 cloudera cloudera 6517 Apr  3 21:29 part-00001.gz
-rwxr-xr-x 1 cloudera cloudera 6525 Apr  3 21:29 part-00002.gz

val x = sc.textFile("mydata_b")
x.count
res2: Long = 10000
```

**Example writing into HDFS**

```
val x = sc.parallelize(List(1,2,3,4,5,6,6,7,9,8,10,21), 3)
x.saveAsTextFile("hdfs://localhost:8020/user/cloudera/test");

val sp = sc.textFile("hdfs://localhost:8020/user/cloudera/sp_data")
sp.flatMap(_.split(" ")).saveAsTextFile("hdfs://localhost:8020/user/cloudera/sp_x")
```

**stats [Double]**

Simultaneously computes the mean, variance and the standard deviation of all values in the RDD.

**Listing Variants**

```
def stats(): StatCounter
```

**Example**

```
val x = sc.parallelize(List(1.0, 2.0, 3.0, 5.0, 20.0, 19.02, 19.29, 11.09, 21.0), 2)
x.stats
res16: org.apache.spark.util.StatCounter = (count: 9, mean: 11.266667, stdev: 8.126859)
```

**sortBy**

This function sorts the input RDD's data and stores it in a new RDD. The first parameter requires you to specify a function ascending or descending order.

**Listing Variants**

```
def sortBy[K](f: (T) => K, ascending: Boolean = true, numPartitions: Int = this.partitions.size)(implicit ord: Order
```

**Example**

```
val y = sc.parallelize(Array(5, 7, 1, 3, 2, 1))
y.sortBy(c => c, true).collect
res101: Array[Int] = Array(1, 1, 2, 3, 5, 7)

y.sortBy(c => c, false).collect
res102: Array[Int] = Array(7, 5, 3, 2, 1, 1)

val z = sc.parallelize(Array(("H", 10), ("A", 26), ("Z", 1), ("L", 5)))
z.sortBy(c => c._1, true).collect
res109: Array[(String, Int)] = Array((A,26), (H,10), (L,5), (Z,1))

z.sortBy(c => c._2, true).collect
res108: Array[(String, Int)] = Array((Z,1), (L,5), (H,10), (A,26))
```

**sortByKey [Ordered]**

This function sorts the input RDD's data and stores it in a new RDD. The output RDD is a shuffled RDD because it stores the data in ranges within the shuffled RDD. Then it sorts these ranges individually with mapPartitions using sort

**Listing Variants**

```
def sortByKey(ascending: Boolean = true, numPartitions: Int = self.partitions.size): RDD[P]
```

**Example**

```
val a = sc.parallelize(List("dog", "cat", "owl", "gnu", "ant"), 2)
val b = sc.parallelize(1 to a.count.toInt, 2)
val c = a.zip(b)
```

```
c.sortByKey(true).collect
res74: Array[(String, Int)] = Array((ant,5), (cat,2), (dog,1), (gnu,4), (owl,3))
c.sortByKey(false).collect
res75: Array[(String, Int)] = Array((owl,3), (gnu,4), (dog,1), (cat,2), (ant,5))

val a = sc.parallelize(1 to 100, 5)
val b = a.cartesian(a)
val c = sc.parallelize(b.takeSample(true, 5, 13), 2)
val d = c.sortByKey(false)
res56: Array[(Int, Int)] = Array((96,9), (84,76), (59,59), (53,65), (52,4))
```

## stdev [Double], sampleStdev [Double]

Calls *stats* and extracts either *stdev*-component or corrected *sampleStdev*-component.

### Listing Variants

```
def stdev(): Double
def sampleStdev(): Double
```

### Example

```
val d = sc.parallelize(List(0.0, 0.0, 0.0), 3)
d.stdev
res10: Double = 0.0
d.sampleStdev
res11: Double = 0.0

val d = sc.parallelize(List(0.0, 1.0), 3)
d.stdev
d.sampleStdev
res18: Double = 0.5
res19: Double = 0.7071067811865476

val d = sc.parallelize(List(0.0, 0.0, 1.0), 3)
d.stdev
res14: Double = 0.4714045207910317
d.sampleStdev
res15: Double = 0.5773502691896257
```

## subtract

Performs the well known standard set subtraction operation:  $A - B$

### Listing Variants

```
def subtract(other: RDD[T]): RDD[T]
def subtract(other: RDD[T], numPartitions: Int): RDD[T]
def subtract(other: RDD[T], p: Partitioner): RDD[T]
```

### Example

```
val a = sc.parallelize(1 to 9, 3)
val b = sc.parallelize(1 to 3, 3)
val c = a.subtract(b)
c.collect
res3: Array[Int] = Array(6, 9, 4, 7, 5, 8)
```

## subtractByKey [Pair]

Very similar to *subtract*, but instead of supplying a function, the key-component of each pair will be automatically used

### Listing Variants

```
def subtractByKey[W: ClassTag](other: RDD[(K, W)]): RDD[(K, V)]
def subtractByKey[W: ClassTag](other: RDD[(K, W)], numPartitions: Int): RDD[(K, V)]
def subtractByKey[W: ClassTag](other: RDD[(K, W)], p: Partitioner): RDD[(K, V)]
```

**Example**

```
val a = sc.parallelize(List("dog", "tiger", "lion", "cat", "spider", "eagle"), 2)
val b = a.keyBy(_.length)
val c = sc.parallelize(List("ant", "falcon", "squid"), 2)
val d = c.keyBy(_.length)
b.subtractByKey(d).collect
res15: Array[(Int, String)] = Array((4,lion))
```

**sum [Double], sumApprox [Double]**

Computes the sum of all values contained in the RDD. The approximate version of the function can finish somewhat fas

**Listing Variants**

```
def sum(): Double
def sumApprox(timeout: Long, confidence: Double = 0.95): PartialResult[BoundedDouble]
```

**Example**

```
val x = sc.parallelize(List(1.0, 2.0, 3.0, 5.0, 20.0, 19.02, 19.29, 11.09, 21.0), 2)
x.sum
res17: Double = 101.39999999999999
```

**take**

Extracts the first  $n$  items of the RDD and returns them as an array. *(Note: This sounds very easy, but it is actually quite c*

**Listing Variants**

```
def take(num: Int): Array[T]
```

**Example**

```
val b = sc.parallelize(List("dog", "cat", "ape", "salmon", "gnu"), 2)
b.take(2)
res18: Array[String] = Array(dog, cat)

val b = sc.parallelize(1 to 10000, 5000)
b.take(100)
res6: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44,
45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68,
69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92,
93, 94, 95, 96, 97, 98, 99, 100)
```

**takeOrdered**

Orders the data items of the RDD using their inherent implicit ordering function and returns the first  $n$  items as an array.

**Listing Variants**

```
def takeOrdered(num: Int)(implicit ord: Ordering[T]): Array[T]
```

**Example**

```
val b = sc.parallelize(List("dog", "cat", "ape", "salmon", "gnu"), 2)
b.takeOrdered(2)
res19: Array[String] = Array(ape, cat)
```

## takeSample

Behaves different from *sample* in the following respects:

- It will return an exact number of samples (*Hint: 2nd parameter*)
- It returns an Array instead of RDD.
- It internally randomizes the order of the items returned.

### Listing Variants

```
def takeSample(withReplacement: Boolean, num: Int, seed: Int): Array[T]
```

### Example

```
val x = sc.parallelize(1 to 1000, 3)
x.takeSample(true, 100, 1)
res3: Array[Int] = Array(339, 718, 810, 105, 71, 268, 333, 360, 341, 300, 68, 848, 431,
449, 773, 172, 802, 339, 431, 285, 937, 301, 167, 69, 330, 864, 40, 645, 65, 349, 613,
468, 982, 314, 160, 675, 232, 794, 577, 571, 805, 317, 136, 860, 522, 45, 628, 178, 321,
482, 657, 114, 332, 728, 901, 290, 175, 876, 227, 130, 863, 773, 559, 301, 694, 460, 839,
952, 664, 851, 260, 729, 823, 880, 792, 964, 614, 821, 683, 364, 80, 875, 813, 951, 663,
344, 546, 918, 436, 451, 397, 670, 756, 512, 391, 70, 213, 896, 123, 858)
```

---

## toDebugString

Returns a string that contains debug information about the RDD and its dependencies.

### Listing Variants

```
def toDebugString: String
```

### Example

```
val a = sc.parallelize(1 to 9, 3)
val b = sc.parallelize(1 to 3, 3)
val c = a.subtract(b)
c.toDebugString
res6: String =
MappedRDD[15] at subtract at <console>:16 (3 partitions)
  SubtractedRDD[14] at subtract at <console>:16 (3 partitions)
    MappedRDD[12] at subtract at <console>:16 (3 partitions)
      ParallelCollectionRDD[10] at parallelize at <console>:12 (3 partitions)
    MappedRDD[13] at subtract at <console>:16 (3 partitions)
      ParallelCollectionRDD[11] at parallelize at <console>:12 (3 partitions)
```

---

## toJavaRDD

Embeds this RDD object within a JavaRDD object and returns it.

### Listing Variants

```
def toJavaRDD(): JavaRDD[T]
```

### Example

```
val c = sc.parallelize(List("Gnu", "Cat", "Rat", "Dog"), 2)
c.toJavaRDD
res3: org.apache.spark.api.java.JavaRDD[String] = ParallelCollectionRDD[6] at
parallelize at <console>:12
```

---

## toLocalIterator

Converts the RDD into a scala iterator at the master node.

#### Listing Variants

```
def toLocalIterator: Iterator[T]
```

#### Example

```
val z = sc.parallelize(List(1,2,3,4,5,6), 2)
val iter = z.toLocalIterator

iter.next
res51: Int = 1

iter.next
res52: Int = 2
```

## top

Utilizes the implicit ordering of `$T$` to determine the top `$k$` values and returns them as an array.

#### Listing Variants

```
def top(num: Int)(implicit ord: Ordering[T]): Array[T]
```

#### Example

```
val c = sc.parallelize(Array(6, 9, 4, 7, 5, 8), 2)
c.top(2)
res28: Array[Int] = Array(9, 8)
```

## toString

Assembles a human-readable textual description of the RDD.

#### Listing Variants

```
override def toString: String
```

#### Example

```
val z = sc.parallelize(List(1,2,3,4,5,6), 2)
z.toString
res61: String = ParallelCollectionRDD[80] at parallelize at <console>:21

val randRDD = sc.parallelize(List( (7,"cat"), (6, "mouse"),(7, "cup"), (6, "book"), (7,
"tv"), (6, "screen"), (7, "heater")))
val sortedRDD = randRDD.sortByKey()
sortedRDD.toString
res64: String = ShuffledRDD[88] at sortByKey at <console>:23
```

## treeAggregate

Computes the same thing as `aggregate`, except it aggregates the elements of the RDD in a multi-level tree pattern. Another parameter is the depth parameter.

#### Listing Variants

```
def treeAggregate[U](zeroValue: U)(seqOp: (U, T) => U, combOp: (U, U) => U, depth: Int = 2)(implicit arg0: ClassTag[U]): RDD[U]
```

#### Example

```
val z = sc.parallelize(List(1,2,3,4,5,6), 2)
```

```
// lets first print out the contents of the RDD with partition labels
def myfunc(index: Int, iter: Iterator[(Int)]) : Iterator[String] = {
  iter.toList.map(x => "[partID:" + index + ", val: " + x + "]").iterator
}

z.mapPartitionsWithIndex(myfunc).collect
res28: Array[String] = Array([partID:0, val: 1], [partID:0, val: 2], [partID:0, val: 3], [partID:1, val: 4],
[partID:1, val: 5], [partID:1, val: 6])

z.treeAggregate(0)(math.max(_, _), _ + _)
res40: Int = 9

// Note unlike normal aggregate. Tree aggregate does not apply the initial value for the second reduce
// This example returns 11 since the initial value is 5
// reduce of partition 0 will be max(5, 1, 2, 3) = 5
// reduce of partition 1 will be max(4, 5, 6) = 6
// final reduce across partitions will be 5 + 6 = 11
// note the final reduce does not include the initial value
z.treeAggregate(5)(math.max(_, _), _ + _)
res42: Int = 11
```

## treeReduce

Works like reduce except reduces the elements of the RDD in a multi-level tree pattern.

### Listing Variants

```
def treeReduce(f: (T, T) => T, depth: Int = 2): T
```

### Example

```
val z = sc.parallelize(List(1,2,3,4,5,6), 2)
z.treeReduce(_+_ )
res49: Int = 21
```

## union, ++

Performs the standard set operation: A union B

### Listing Variants

```
def ++(other: RDD[T]): RDD[T]
def union(other: RDD[T]): RDD[T]
```

### Example

```
val a = sc.parallelize(1 to 3, 1)
val b = sc.parallelize(5 to 7, 1)
(a ++ b).collect
res0: Array[Int] = Array(1, 2, 3, 5, 6, 7)
```

## unpersist

Dematerializes the RDD (*i.e. Erases all data items from hard-disk and memory*). However, the RDD object remains. If i

### Listing Variants

```
def unpersist(blocking: Boolean = true): RDD[T]
```

### Example

```
val y = sc.parallelize(1 to 10, 10)
val z = (y++y)
z.collect
```

```
z.unpersist(true)
14/04/19 03:04:57 INFO UnionRDD: Removing RDD 22 from persistence list
14/04/19 03:04:57 INFO BlockManager: Removing RDD 22
```

## values

Extracts the values from all contained tuples and returns them in a new RDD.

### Listing Variants

```
def values: RDD[V]
```

### Example

```
val a = sc.parallelize(List("dog", "tiger", "lion", "cat", "panther", "eagle"), 2)
val b = a.map(x => (x.length, x))
b.values.collect
res3: Array[String] = Array(dog, tiger, lion, cat, panther, eagle)
```

## variance [Double], sampleVariance [Double]

Calls stats and extracts either *variance*-component or corrected *sampleVariance*-component.

### Listing Variants

```
def variance(): Double
def sampleVariance(): Double
```

### Example

```
val a = sc.parallelize(List(9.1, 1.0, 1.2, 2.1, 1.3, 5.0, 2.0, 2.1, 7.4, 7.5, 7.6, 8.8, 10.0, 8.9, 5.5), 3)
a.variance
res70: Double = 10.605333333333332

val x = sc.parallelize(List(1.0, 2.0, 3.0, 5.0, 20.0, 19.02, 19.29, 11.09, 21.0), 2)
x.variance
res14: Double = 66.04584444444443

x.sampleVariance
res13: Double = 74.30157499999999
```

## zip

Joins two RDDs by combining the i-th of either partition with each other. The resulting RDD will consist of two-compo

### Listing Variants

```
def zip[U: ClassTag](other: RDD[U]): RDD[(T, U)]
```

### Example

```
val a = sc.parallelize(1 to 100, 3)
val b = sc.parallelize(101 to 200, 3)
a.zip(b).collect
res1: Array[(Int, Int)] = Array((1,101), (2,102), (3,103), (4,104), (5,105), (6,106), (7,107), (8,108), (9,109), (10,110), (11,111), (12,112), (13,113), (14,114), (15,115), (16,116), (17,117), (18,118), (19,119), (20,120), (21,121), (22,122), (23,123), (24,124), (25,125), (26,126), (27,127), (28,128), (29,129), (30,130), (31,131), (32,132), (33,133), (34,134), (35,135), (36,136), (37,137), (38,138), (39,139), (40,140), (41,141), (42,142), (43,143), (44,144), (45,145), (46,146), (47,147), (48,148), (49,149), (50,150), (51,151), (52,152), (53,153), (54,154), (55,155), (56,156), (57,157), (58,158), (59,159), (60,160), (61,161), (62,162), (63,163), (64,164), (65,165), (66,166), (67,167), (68,168), (69,169), (70,170), (71,171), (72,172), (73,173), (74,174), (75,175), (76,176), (77,177), (78,...)

val a = sc.parallelize(1 to 100, 3)
```

```
val b = sc.parallelize(101 to 200, 3)
val c = sc.parallelize(201 to 300, 3)
a.zip(b).zip(c).map((x) => (x._1._1, x._1._2, x._2)).collect
res12: Array[(Int, Int, Int)] = Array((1,101,201), (2,102,202), (3,103,203), (4,104,204),
(5,105,205), (6,106,206), (7,107,207), (8,108,208), (9,109,209), (10,110,210),
(11,111,211), (12,112,212), (13,113,213), (14,114,214), (15,115,215), (16,116,216),
(17,117,217), (18,118,218), (19,119,219), (20,120,220), (21,121,221), (22,122,222),
(23,123,223), (24,124,224), (25,125,225), (26,126,226), (27,127,227), (28,128,228),
(29,129,229), (30,130,230), (31,131,231), (32,132,232), (33,133,233), (34,134,234),
(35,135,235), (36,136,236), (37,137,237), (38,138,238), (39,139,239), (40,140,240),
(41,141,241), (42,142,242), (43,143,243), (44,144,244), (45,145,245), (46,146,246),
(47,147,247), (48,148,248), (49,149,249), (50,150,250), (51,151,251), (52,152,252),
(53,153,253), (54,154,254), (55,155,255)...
```

## zipPartitions

Similar to *zip*. But provides more control over the zipping process.

### Listing Variants

```
def zipPartitions[B: ClassTag, V: ClassTag](rdd2: RDD[B])(f: (Iterator[T], Iterator[B]) => Iterator[V]): RDD[V]
def zipPartitions[B: ClassTag, V: ClassTag](rdd2: RDD[B], preservesPartitioning: Boolean)(f: (Iterator[T], Iterator[B]) => Iterator[V]): RDD[V]
def zipPartitions[B: ClassTag, C: ClassTag, V: ClassTag](rdd2: RDD[B], rdd3: RDD[C])(f: (Iterator[T], Iterator[B], Iterator[C]) => Iterator[V]): RDD[V]
def zipPartitions[B: ClassTag, C: ClassTag, V: ClassTag](rdd2: RDD[B], rdd3: RDD[C], preservesPartitioning: Boolean)(f: (Iterator[T], Iterator[B], Iterator[C]) => Iterator[V]): RDD[V]
def zipPartitions[B: ClassTag, C: ClassTag, D: ClassTag, V: ClassTag](rdd2: RDD[B], rdd3: RDD[C], rdd4: RDD[D])(f: (Iterator[T], Iterator[B], Iterator[C], Iterator[D]) => Iterator[V]): RDD[V]
def zipPartitions[B: ClassTag, C: ClassTag, D: ClassTag, V: ClassTag](rdd2: RDD[B], rdd3: RDD[C], rdd4: RDD[D], preservesPartitioning: Boolean)(f: (Iterator[T], Iterator[B], Iterator[C], Iterator[D]) => Iterator[V]): RDD[V]
```

### Example

```
val a = sc.parallelize(0 to 9, 3)
val b = sc.parallelize(10 to 19, 3)
val c = sc.parallelize(100 to 109, 3)
def myfunc(aiter: Iterator[Int], biter: Iterator[Int], citer: Iterator[Int]): Iterator[String] =
{
  var res = List[String]()
  while (aiter.hasNext && biter.hasNext && citer.hasNext)
  {
    val x = aiter.next + " " + biter.next + " " + citer.next
    res ::= x
  }
  res.iterator
}
a.zipPartitions(b, c)(myfunc).collect
res50: Array[String] = Array(2 12 102, 1 11 101, 0 10 100, 5 15 105, 4 14 104, 3 13 103,
9 19 109, 8 18 108, 7 17 107, 6 16 106)
```

## zipWithIndex

Zips the elements of the RDD with its element indexes. The indexes start from 0. If the RDD is spread across multiple partitions, the indexes are local to each partition.

### Listing Variants

```
def zipWithIndex(): RDD[(T, Long)]
```

### Example

```
val z = sc.parallelize(Array("A", "B", "C", "D"))
val r = z.zipWithIndex
res110: Array[(String, Long)] = Array((A,0), (B,1), (C,2), (D,3))

val z = sc.parallelize(100 to 120, 5)
val r = z.zipWithIndex
r.collect
res11: Array[(Int, Long)] = Array((100,0), (101,1), (102,2), (103,3), (104,4), (105,5), (106,6),
(107,7), (108,8), (109,9), (110,10), (111,11), (112,12), (113,13), (114,14), (115,15), (116,16),
(117,17), (118,18), (119,19), (120,20))
```

## zipWithUniqueId

This is different from `zipWithIndex` since it just gives a unique id to each data element but the ids may not match the index. Compare the results of the example below with that of the 2nd example of `zipWithIndex`. You should be able to see the difference.

### Listing Variants

```
def zipWithUniqueId(): RDD[(T, Long)]
```

### Example

```
val z = sc.parallelize(100 to 120, 5)
val r = z.zipWithUniqueId
r.collect

res12: Array[(Int, Long)] = Array((100,0), (101,5), (102,10), (103,15), (104,1), (105,6), (106,11),
(107,16), (108,2), (109,7), (110,12), (111,17), (112,3), (113,8), (114,13), (115,18), (116,4), (117,9),
(118,14), (119,19), (120,24))
```

  
[hit counter website](#)