Apache Spark API By Example

A Command Reference for Beginners

Matthias Langer, Zhen He
Department of Computer Science and Computer Engineering
La Trobe University
Bundoora, VIC 3086
Australia
m.langer@latrobe.edu.au, z.he@latrobe.edu.au

May 31, 2014

Contents

1 Preface

Spark is an advanced open-source cluster computing system that is capable of handling extremely large data sets. It was first published by **?** and its popularity has increased ever since. Due to its real-time properties and efficient usage of resources, Spark has become a very popular alternative to well established computational software for big data.

Spark is still actively being maintained and further developed by its original creators from UC Berkeley. Hence, this command reference and the associated, including the code-snippets and sample outputs outputs shown, should be considered as a overview of the status-quo of this amazing piece of software technology. Specifically, the API examples in this document are for **Spark version 0.9**. However, we do not expect the API to change much in future releases.

This document does not cover any installation or distribution related topics. For installation instructions, please refer to the Apache Spark website.

2 Shell Configuration

One of the strongest features of Spark is its shell. The Spark-Shell allows users to type and execute commands in a Unix-Terminal-like fashion. The preferred language to use is probably Scala, which is actually a heavily modified Java dialect that enhances the language with many features and concepts of functional programming languages. Below are just a few of the more useful Spark-Shell configuration parameters.

2.1 Adjusting the amount of memory

To adjust the amount of memory that Spark may use for executing queries you have to set the following environment prior to starting the shell *(Hint: If you add this command to the end of your .bashrc-file, the environment variable will be set automatically every time when you open a terminal)*:

```
export SPARK_MEM=1g
```

In the above example we are setting the maximum amount of memory to 1 GB.

2.2 Adjusting the number of worker threads

The following environment variable controls the number of worker threads that Spark uses: *(Hint: If you add this command to the end of your .bashrc-file, the environment variable will be set automatically every time when you open a terminal)*:

```
export SPARK_WORKER_INSTANCES=4
```

If you run Spark in local mode you can also set the number of worker threads in one setting as follows:

```
export MASTER=local[32]
```

2.3 Adding a Listener to the Logging System

The Spark-Shell is able to print logs automatically. It is possible to manually specify a directory for log-files. However, if no directory is set, */tmp* will be used instead. The following snippet shows how one can setup a log-writer manually:

```
export SPARK_LOG_DIR=/home/cloudera/Documents/mylog
sh spark-shell
scala> var logger = new org.apache.spark.scheduler.JobLogger(
       "cloudera", "someLogDirName")
scala> sc.addSparkListener(logger)
```

3 The RDD API

RDD is short for *Resilient Distributed Dataset*. RDDs are the workhorse of the Spark system. As a user, one can consider a RDD as a handle for a collection of individual data partitions which are the result of some computation.

However, an RDD is actually more than that. On cluster installations, separate data partitions can be on separate nodes. Using the RDD as a handle one can access all partitions and perform computations and transformations using the contained data. Whenever a part of a RDD or an entire RDD is lost, the system is able to reconstruct the data of lost partitions by using lineage information. Lineage refers to the sequence of transformations used to produce the current RDD. As a result, Spark is able to recover automatically from most failures.

All RDDs available in Spark derive either directly or indirectly from the class RDD. This class comes with a large set of methods that perform operations on the data within the associated partitions. The class RDD is abstract. Whenever, one uses a RDD, one is actually using a concertized implementation of RDD. These implementations have to overwrite some core functions to make the RDD behave as expected.

One reason why Spark has lately become a very popular system for processing big data is that it does not impose restrictions regarding what data can be stored within RDD partitions. The RDD API already contains many useful operations. But, because the creators of Spark had to keep 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. Therefore Spark extended the interface of RDD to provide additional functions (PairRDDFunctions) which explicitly work on key-value pairs. Currently, 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 of an RDD are implicitly convertible to the Scala data-type *double*.

PairRDDFunctions Methods defined in this interface extension become available when the data items have a two component tuple structure. Spark will interpret the first tuple item *(i.e. tuplename._1)* as the key and the second item *(i.e. tuplename._2)* as the associated value.

OrderedRDDFunctions Methods defined in this interface extension become available if the data items are two component tuples where the key is implicitly sortable.

SequenceFileRDDFunctions This extension contains several methods that allow users to create Hadoop sequence-files from RDDs. The data items must be two component key-value tuples as required by the PairRDDFunctions. However, there are additional requirements considering the convertibility of the tuple components to *Writable* types.

Since Spark will make methods with extended functionality automatically available to users when the data items fulfill the above described requirements, we decided to list all possible available functions in strictly alphabetical order. We will append either of the following flags [Double] , [Ordered], [Pair] or [SeqFile]to the function-name to indicate it belongs to an extension that requires the data items to conform to a certain format or type.

3.1 aggregate

The *aggregate*-method provides an interface for performing highly customized reductions and aggregations with a RDD. However, due to the way Scala and Spark execute and process data, care must be taken to achieve deterministic behavior. The following list contains a few observations we made while experimenting with *aggregate*:

- The reduce and combine functions have to be commutative and associative.
- As can be seen from the function definition below, the output of the combiner must be equal to its input. This is necessary because Spark will chain-execute it.
- The zero value is the initial value of the U component when either *seqOp* or *combOp* are executed for the first element of their domain of influence. Depending on what you want to achieve, you may have to change it. However, to make your code deterministic, make sure that your code will yield the same result regardless of the number or size of partitions.
- Do not assume any execution order for either partition computations or combining partitions.
- The neutral *zeroValue* is applied at the beginning of each sequence of reduces within the individual partitions and again when the output of separate partitions is combined.
- Why have two separate combine functions? The first functions maps the input values into the result space. Note that the aggregation data type (1^{st} input and output) can be different ($U \neq T$). The second function reduces these mapped values in the result space.
- Why would one want to use two input data types? Let us assume we do an archaeological site survey using a metal detector. While walking through the site we take GPS coordinates of important findings based on the output of the metal

detector. Later, we intend to draw an image of a map that highlights these locations using the *aggregate* function. In this case the *zeroValue* could be an area map with no highlights. The possibly huge set of input data is stored as GPS coordinates across many partitions. *seqOp* could convert the GPS coordinates to map coordinates and put a marker on the map at the respective position. *combOp* will receive these highlights as partial maps and combine them into a single final output map.

Listing 3.1: Variants

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

Listing 3.2: Examples

```
val z = sc.parallelize(List(1,2,3,4,5,6), 2)
z.aggregate(0)(math.max(_, _), _ + _)
res40: Int = 9

val z = sc.parallelize(List("a","b","c","d","e","f"),2)
z.aggregate("")(_ + _, _+_)
res115: String = abcdef

z.aggregate("x")(_ + _, _+_)
res116: String = xxdefxabc

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 recursively reducing any further within the partition for the final string.

Listing 3.3: 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 results in length of zero being input to the second reduce which then upgrades it a length of 1. *(Warning: The above example shows bad design since the output is dependent on the order of the data inside the partitions.)*

3.2 cartesian

Computes the cartesian product between two RDDs (i.e. Each item of the first RDD is joined with each item of the second RDD) and returns them as a new RDD. *(Warning: Be careful when using this function.! Memory consumption can quickly become an issue!)*

Listing 3.4: Variants

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

Listing 3.5: Examples

```
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))
```

3.3 checkpoint

Will create a checkpoint when the RDD is computed next. Checkpointed RDDs are stored as a binary file within the checkpoint directory which can be specified using the Spark context. *(Warning: Spark applies lazy evaluation. Checkpointing will not occur until an action is invoked.)*

Important note: the directory "my_directory_name" should exist in all slaves. As an alternative you could use an HDFS directory URL as well.

Listing 3.6: Variants

```
def checkpoint()
```

Listing 3.7: Examples

```
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
```

3.4 coalesce, repartition

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

Listing 3.8: Variants

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

Listing 3.9: Examples

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

3.5 cogroup[Pair] , groupWith[Pair]

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

Listing 3.10: Variants

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

Listing 3.11: 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, (Seq[String], Seq[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, (Seq[String], Seq[String], Seq[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, (Seq[String], Seq[String]))] = Array(
(4,(ArrayBuffer(kiwi),ArrayBuffer(iPad))),
(2,(ArrayBuffer(banana),ArrayBuffer())),
(3,(ArrayBuffer(orange),ArrayBuffer())),
(1,(ArrayBuffer(apple),ArrayBuffer(laptop, desktop))),
(5,(ArrayBuffer(),ArrayBuffer(computer))))
```

3.6 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 applied before inserting the values into the result array.

Listing 3.12: Variants

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

Listing 3.13: Examples

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

3.7 collectAsMap[Pair]

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

Listing 3.14: Variants

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

Listing 3.15: 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)
```

3.8 combineByKey[Pair]

Very efficient implementation that combines the values of a RDD consisting of two-component tuples by applying multiple aggregators one after another.

Listing 3.16: Variants

```
def combineByKey[C](createCombiner: V => C, mergeValue: (C, V) => C,
    mergeCombiners: (C, C) => C): RDD[(K, C)]
def combineByKey[C](createCombiner: V => C, mergeValue: (C, V) => C,
    mergeCombiners: (C, C) => C, numPartitions: Int): RDD[(K, C)]
def combineByKey[C](createCombiner: V => C, mergeValue: (C, V) => C,
    mergeCombiners: (C, C) => C, partitioner: Partitioner,
    mapSideCombine: Boolean = true, serializerClass: String = null):
    RDD[(K, C)]
```

Listing 3.17: Examples

```
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)))
```

3.9 compute

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

Listing 3.18: Variants

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

3.10 context, sparkContext

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

Listing 3.19: Variants

```
def context
def sparkContext: SparkContext
```

Listing 3.20: Examples

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

3.11 count

Returns the number of items stored within a RDD.

Listing 3.21: Variants

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

Listing 3.22: Examples

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

3.12 countApprox

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

Listing 3.23: Variants

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

3.13 countByKey[Pair]

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

Listing 3.24: Variants

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

Listing 3.25: Examples

```
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)
```

3.14 countByKeyApprox[Pair]

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

Listing 3.26: Variants

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

3.15 countByValue

Returns a map that contains all unique values of the RDD and their respective occurrence counts. *(Warning: This operation will finally aggregate the information in a single reducer!)*

Listing 3.27: Variants

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

Listing 3.28: Examples

```
val b = sc.parallelize(List(1,2,3,4,5,6,7,8,2,4,2,1,1,1,1,1))
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)
```

3.16 countByValueApprox

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

Listing 3.29: Variants

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

3.17 countApproxDistinct

Computes the approximate number of distinct values. For large RDDs which are spread across many nodes, this function may execute faster than other counting methods. The parameter *relativeSD* controls the accuracy of the computation.

Listing 3.30: Variants

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

Listing 3.31: Examples

```
val a = sc.parallelize(1 to 10000, 20)
val b = a++a++a++a++a
b.countApproxDistinct(0.1)
res14: Long = 10784

b.countApproxDistinct(0.05)
res15: Long = 11055

b.countApproxDistinct(0.01)
res16: Long = 10040

b.countApproxDistinct(0.001)
res0: Long = 10001
```

3.18 countApproxDistinctByKey[Pair]

Similar to *countApproxDistinct*, but computes the approximate number of distinct values for each distinct key. Hence, the RDD must consist of two-component tuples. For large RDDs which are spread across many nodes, this function may execute faster than other counting methods. The parameter *relativeSD* controls the accuracy of the computation.

Listing 3.32: 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)]
```

Listing 3.33: Examples

```
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))
```

3.19 dependencies

Returns the RDD on which this RDD depends.

Listing 3.34: Variants

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

Listing 3.35: Examples

```
val b = sc.parallelize(List(1,2,3,4,5,6,7,8,2,4,2,1,1,1,1,1))
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)
```

3.20 distinct

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

Listing 3.36: Variants

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

Listing 3.37: Examples

```
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
```

3.21 first

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

Listing 3.38: Variants

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

Listing 3.39: Examples

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

3.22 filter

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

Listing 3.40: Variants

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

Listing 3.41: Examples

```
val a = sc.parallelize(1 to 10, 3)
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 partial functions to deal with mixed data-types. *(Tip: Partial functions are very useful if you have some data which may be bad and you do not want to handle but for the good data (matching data) you want to apply some kind of map function. The following article is good. It teaches you about partial functions in a very nice way and explains why case has to be used for partial functions: http://blog.bruchez.name/2011/10/scala-partial-functions-without-phd.html)*

Listing 3.42: 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* property of a function-object to determine whether the test-function is compatible with each data item. Only data items that pass this test *(=filter)* are then mapped using the function-object.

Listing 3.43: 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 explicitly declare what type you want instead of any. Otherwise the compiler does (apparently) not know what bytecode it should produce:

Listing 3.44: Examples

```
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>
```

3.23 filterWith

This is an extended version of *filter*. It takes two function arguments. The first argument must conform to $Int \Rightarrow T$ and is executed once per partition. It will transform the partition index to type T. The second function looks like $(U, T) \Rightarrow \textbf{Boolean}$. T is the transformed partition 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 3.45: Variants

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

Listing 3.46: 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)
```

3.24 flatMap

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

Listing 3.47: Variants

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

Listing 3.48: Examples

```
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, 4, 5,
    5, 6, 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)
```

3.25 flatMapValues[Pair]

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

Listing 3.49: Variants

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

Listing 3.50: Examples

```
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,c), (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))
```

3.26 flatMapWith

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

Listing 3.51: Variants

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

Listing 3.52: Examples

```
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)
```

3.27 fold

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

Listing 3.53: Variants

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

Listing 3.54: Examples

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

3.28 foldByKey[Pair]

Very similar to *fold*, but performs the folding separately for each key of the RDD. This function is only available if the RDD consists of two-component tuples.

Listing 3.55: 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)]
```

Listing 3.56: Examples

```
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))
```

3.29 foreach

Executes an parameterless function for each data item.

Listing 3.57: Variants

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

Listing 3.58: Examples

```
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
```

3.30 foreachPartition

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

Listing 3.59: Variants

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

Listing 3.60: Examples

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

3.31 foreachWith

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

Listing 3.61: Variants

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

Listing 3.62: Examples

```
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
```

3.32 generator, setGenerator

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

Listing 3.63: Variants

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

3.33 getCheckpointFile

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

Listing 3.64: Variants

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

Listing 3.65: Examples

```
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)
```

3.34 preferredLocations

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

Listing 3.66: Variants

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

3.35 getStorageLevel

Retrieves the currently set storage level of the RDD. This can only be used to assign a new storage level if the RDD does not have a storage level set yet. The example below shows the error you will get, when you try to reassign the storage level.

Listing 3.67: Variants

```
def getStorageLevel
```

Listing 3.68: Examples

```
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
```

3.36 glom

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

Listing 3.69: Variants

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

Listing 3.70: Examples

```
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))
```

3.37 groupBy

Listing 3.71: Variants

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

Listing 3.72: Examples

```
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))
    , (odd,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))))
```

3.38 groupByKey[Pair]

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

Listing 3.73: Variants

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

Listing 3.74: Examples

```
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)))
```

3.39 histogram[Double]

These functions take an RDD of doubles and create a histogram with either even spacing (the number of buckets equals to *bucketCount*) or arbitrary spacing based on custom bucket boundaries supplied by the user via an array of double values. The result type of both variants is slightly different, the first function will return a tuple consisting of two arrays. The first array contains the computed bucket boundary values and the second array contains the corresponding count of values *(i.e. the histogram)*. The second variant of the function will just return the histogram as an array of integers.

Listing 3.75: Variants

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

Listing 3.76: Examples 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))
```

Listing 3.77: 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)
```

3.40 id

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

Listing 3.78: Variants

```
val id: Int
```

Listing 3.79: Examples

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

3.41 isCheckpointed

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

Listing 3.80: Variants

```
def isCheckpointed: Boolean
```

Listing 3.81: Examples

```
sc.setCheckpointDir("/home/cloudera/Documents")
c.isCheckpointed
res6: Boolean = false
c.checkpoint
c.isCheckpointed
res8: Boolean = false
c.collect
c.isCheckpointed
res9: Boolean = true
```

3.42 iterator

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

Listing 3.82: Variants

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

3.43 join[Pair]

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

Listing 3.83: 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))]
```

Listing 3.84: Examples

```
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

res17: Array[(Int, (String, String))] = Array((6,(salmon,salmon)), (6,(
    salmon,rabbit)), (6,(salmon,turkey)), (6,(rabbit,salmon)), (6,(
    rabbit,rabbit)), (6,(rabbit,turkey)), (6,(turkey,salmon)), (6,(
    turkey,rabbit)), (6,(turkey,turkey)), (3,(dog,dog)), (3,(dog,cat)),
     (3,(dog,gnu)), (3,(dog,bee)), (3,(cat,dog)), (3,(cat,cat)), (3,(
    cat,gnu)), (3,(cat,bee)), (3,(gnu,dog)), (3,(gnu,cat)), (3,(gnu,gnu
    )), (3,(gnu,bee)), (3,(bee,dog)), (3,(bee,cat)), (3,(bee,gnu)),
    (3,(bee,bee)), (4,(wolf,wolf)), (4,(wolf,bear)), (4,(bear,wolf)),
    (4,(bear,bear)))
```

3.44 keyBy

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

Listing 3.85: Variants

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

Listing 3.86: Examples

```
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))
```

3.45 keys[Pair]

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

Listing 3.87: Variants

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

Listing 3.88: Examples

```
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)
```

3.46 leftOuterJoin[Pair]

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

Listing 3.89: 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]))]
```

Listing 3.90: Examples

```
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)))
```

3.47 lookup[Pair]

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

Listing 3.91: Variants

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

Listing 3.92: Examples

```
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)
```

3.48 map

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

Listing 3.93: Variants

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

Listing 3.94: Examples

```
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))
```

3.49 mapPartitions

This is a specialized map that is called only once for each partition. The entire content of the respective partitions is available as a sequential stream of values via the input argument (*Iterarator[T]*). The custom function must return yet another *Iterator[U]*. The combined result iterators are automatically converted into a new RDD. Please note, that the tuples (3,4) and (6,7) are missing from the following result due to the partitioning we chose.

Listing 3.95: Variants

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

Listing 3.96: Examples

```
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))
```

Listing 3.97: Example 2

```
val x = sc.parallelize(List("1", "2", "3", "4", "5", "6", "7", "8",
    "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, 4,
    4, 4, 4, 4, 4, 4, 5, 7, 7, 7, 9, 9, 10)
```

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

Listing 3.98: Example 2 with 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, 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)
```

3.50 mapPartitionsWithContext

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

Listing 3.99: Variants

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

Listing 3.100: Examples

```
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   +
    ", Interrupted: " + tc.interrupted))
```

```
  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)
Partition: 1, AttemptID: 1, Interrupted: false
...
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)
Partition: 2, AttemptID: 2, Interrupted: false
?
res0: Array[Int] = Array(2, 6, 4, 8)
```

3.51 mapPartitionsWithIndex

Similar to *mapPartitions*, but takes two parameters. The first parameter is the index of the partition and the second is an iterator through all the items within this partition. The output is an iterator containing the list of items after applying whatever transformation the function encodes.

Listing 3.101: Variants

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

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)
```

3.52 mapPartitionsWithSplit

This method has been marked as deprecated in the API. So, you should not use this method anymore. Deprecated methods will not be covered in this document.

Listing 3.102: Variants

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

3.53 mapValues[Pair]

Takes the values of a RDD that consists of two-component tuples, and applies the provided function to transform each value. Then, it forms new two-component tuples using the key and the transformed value and stores them in a new RDD.

Listing 3.103: Variants

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

Listing 3.104: Examples

```
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))
```

3.54 mapWith

This is an extended version of *map*. It takes two function arguments. The first argument must conform to $Int \Rightarrow T$ and is executed once per partition. It will map the partition index to some transformed partition index of type T. The second function must conform to $(U, T) \Rightarrow U$. T is the transformed partition index and U is a data item of the RDD. Finally the function has to return a transformed data item of type U.

Listing 3.105: Variants

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

Listing 3.106: Examples

```
val x = sc.parallelize(List(1,2,3,4,5,6,7,8,9,10), 3)
x.mapWith(a => a * 10)((a, b) => (b + 2)).collect
res4: Array[Int] = Array(2, 2, 2, 12, 12, 12, 22, 22, 22, 22)

val x = sc.parallelize(List(1,2,3,4,5,6,7,8,9,10), 3)
x.mapWith(a => a * 10)((a, b) => (a + 2)).collect
res5: Array[Int] = Array(3, 4, 5, 6, 7, 8, 9, 10, 11, 12)

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)
```

3.55 mean[Double] , meanApprox[Double]

Calls *stats* and extracts the mean component. The approximate version of the function can finish somewhat faster in some scenarios. However, it trades accuracy for speed.

Listing 3.107: Variants

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

Listing 3.108: Examples

```
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
```

3.56 name, setName

Allows a RDD to be tagged with a custom name.

Listing 3.109: Variants

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

Listing 3.110: Examples

```
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
```

3.57 partitionBy[Pair]

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

Listing 3.111: Variants

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

3.58 partitioner

Specifies a function pointer to the default partitioner that will be used for *groupBy*, *subtract*, *reduceByKey* (from **PairedRDDFunctions**), etc. functions.

Listing 3.112: Variants

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

3.59 partitions

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

Listing 3.113: Variants

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

Listing 3.114: Examples

```
b.partitions
res1: Array[org.apache.spark.Partition] = Array(org.apache.spark.rdd.
    ParallelCollectionPartition@691, org.apache.spark.rdd.
    ParallelCollectionPartition@692, org.apache.spark.rdd.
    ParallelCollectionPartition@693)
```

3.60 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 identifier to decide which partitions should be kept. The parameterless variants *persist()* and *cache()* are just abbreviations for *persist(StorageLevel.MEMORY_ONLY). (Warning: Once the storage level has been changed, it cannot be changed again!)*

Listing 3.115: Variants

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

Listing 3.116: Examples

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

3.61 pipe

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

Listing 3.117: 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 = null, printRDDElement:
     (T, String => Unit) => Unit = null): RDD[String]
```

Listing 3.118: Examples

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

3.62 reduce

This function provides the well-known *reduce* functionality in Spark. Please note that any function f you provide, should be commutative in order to generate reproducible results.

Listing 3.119: Variants

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

Listing 3.120: Examples

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

3.63 reduceByKey[Pair], reduceByKeyLocally[Pair], reduceByKeyToDriver[Pair]

Very similar to *reduce*, but performs the reduction separately for each key of the RDD. This function is only available if the RDD consists of two-component tuples.

Listing 3.121: 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]
```

Listing 3.122: Examples

```
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))
```

3.64 rightOuterJoin[Pair]

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

Listing 3.123: 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))]
```

Listing 3.124: Examples

```
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)))
```

3.65 sample

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

Listing 3.125: Variants

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

Listing 3.126: Examples

```
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
```

3.66 saveAsHadoopFile[Pair] , saveAsHadoopDataset[Pair] , saveAsNewAPIHadoopFile[Pair]

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

Listing 3.127: 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 fm: ClassTag[F])
def saveAsHadoopFile(path: String, keyClass: Class[_], valueClass:
    Class[_], outputFormatClass: Class[_ <: OutputFormat[_, _]], codec:
     Class[_ <: CompressionCodec])
def saveAsHadoopFile(path: String, keyClass: Class[_], valueClass:
    Class[_], outputFormatClass: Class[_ <: OutputFormat[_, _]], conf:
    JobConf = new JobConf(self.context.hadoopConfiguration), codec:
    Option[Class[_ <: CompressionCodec]] = None)
def saveAsNewAPIHadoopFile[F <: NewOutputFormat[K, V]](path: String)(
    implicit fm: ClassTag[F])
def saveAsNewAPIHadoopFile(path: String, keyClass: Class[_], valueClass
    : Class[_], outputFormatClass: Class[_ <: NewOutputFormat[_, _]],
    conf: Configuration = self.context.hadoopConfiguration)
```

3.67 saveAsObjectFile

Saves the RDD in binary format.

Listing 3.128: Variants

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

Listing 3.129: Examples

```
val x = sc.parallelize(1 to 100, 3)
x.saveAsObjectFile("objFile")
val y = sc.objectFile[Array[Int]]("objFile")
y.collect
```

```
res52: Array[Int] = 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, 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, 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)
```

3.68 saveAsSequenceFile[SeqFile]

Saves the RDD as a Hadoop sequence file.

Listing 3.130: Variants

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

Listing 3.131: Examples

```
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
```

3.69 saveAsTextFile

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

Listing 3.132: Variants

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

Listing 3.133: Examples 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
```

Listing 3.134: Examples 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
```

Listing 3.135: 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")
```

3.70 stats[Double]

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

Listing 3.136: Variants

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

Listing 3.137: Examples

```
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)
```

3.71 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 data that is output by a reducer which has been shuffled. The implementation of this function is actually very clever. First, it uses a range partitioner to partition the data in ranges within the shuffled RDD. Then it sorts these ranges individually with mapPartitions using standard sort mechanisms.

Listing 3.138: Variants

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

Listing 3.139: Examples

```
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))
```

3.72 stdev[Double] , sampleStdev[Double]

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

$$\text{stdev} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \overline{x})^2} \tag{3.1}$$

$$\text{sampleStdev} = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \overline{x})^2} \tag{3.2}$$

Listing 3.140: Variants

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

Listing 3.141: Examples

```
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
```

3.73 subtract

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

Listing 3.142: 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]
```

Listing 3.143: Examples

```
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)
```

3.74 subtractByKey[Pair]

Very similar to *subtract*, but instead of supplying a function, the key-component of each pair will be automatically used as criterion for removing items from the first RDD.

Listing 3.144: 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)]
```

Listing 3.145: Examples

```
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))
```

3.75 sum[Double] , sumApprox[Double]

Computes the sum of all values contained in the RDD. The approximate version of the function can finish somewhat faster in some scenarios. However, it trades accuracy for speed.

Listing 3.146: Variants

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

Listing 3.147: Examples

```
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
```

3.76 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 a tricky problem for the implementors of Spark because the items in question can be in many different partitions.)*

Listing 3.148: Variants

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

Listing 3.149: Examples

```
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)
```

3.77 takeOrdered

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

Listing 3.150: Variants

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

Listing 3.151: Examples

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

3.78 takeSample

Behaves different from *sample* in the following respects:

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

Listing 3.152: Variants

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

Listing 3.153: Examples

```
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)
```

3.79 toDebugString

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

Listing 3.154: Variants

```
def toDebugString: String
```

Listing 3.155: Examples

```
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)
```

3.80 toJavaRDD

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

Listing 3.156: Variants

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

Listing 3.157: Examples

```
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
```

3.81 top

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

Listing 3.158: Variants

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

Listing 3.159: Examples

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

3.82 toString

Assembles a human-readable textual description of the RDD.

Listing 3.160: Variants

```
override def toString: String
```

Listing 3.161: Examples

```
val a = sc.parallelize(1 to 9, 3)
val b = sc.parallelize(1 to 3, 3)
val c = a.subtract(b)
c.toString
res7: String = MappedRDD[15] at subtract at <console>:16
```

3.83 union, ++

Performs the standard set operation: $A \cup B$

Listing 3.162: Variants

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

Listing 3.163: Examples

```
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)
```

3.84 unpersist

Dematerializes the RDD *(i.e. Erases all data items from hard-disk and memory).* However, the RDD object remains. If it is referenced in a computation, Spark will regenerate it automatically using the stored dependency graph.

Listing 3.164: Variants

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

Listing 3.165: Examples

```
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
```

3.85 values[Pair]

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

Listing 3.166: Variants

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

Listing 3.167: Examples

```
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)
```

3.86 variance[Double] , sampleVariance[Double]

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

Listing 3.168: Variants

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

Listing 3.169: Examples

```
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
```

3.87 zip

Joins two RDDs by combining the i-th of either partition with each other. The resulting RDD will consist of two-component tuples which are interpreted as key-value pairs by the methods provided by the PairRDDFunctions extension.

Listing 3.170: Variants

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

Listing 3.171: Examples

```
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)...
```

3.88 zipParititions

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

Listing 3.172: 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]
```

Listing 3.173: Examples

```
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)
```

4 Further Topics

4.1 Reading from HDFS

This requires you to upload your data first into HDFS using whatever method you prefer.

Listing 4.1: Examples

```
val sp = sc.textFile("hdfs://localhost:8020/user/cloudera/sp_data")
sp.toDebugString
res25: String =
MappedRDD[32] at textFile at <console>:12 (1 partitions)
HadoopRDD[31] at textFile at <console>:12 (1 partition
sp.collect
res24: Array[String] = Array(A MIDSUMMER-NIGHT'S DREAM, "", "Now , fair
     Hippolyta , our nuptial hour ", "Draws on apace : four happy days
    bring in ", "Another moon ; but O ! methinks how slow ", This old
    moon wanes ; she lingers my desires ,, "Like to a step dame , or a
    dowager ", Long withering out a young man's revenue ., "", Four
    days will quickly steep themselves in night ;, Four nights will
    quickly dream away the time ;, "And then the moon , like to a
    silver bow ", "New-bent in heaven , shall behold the night ", Of
    our solemnities ., "", Go , Philostrate ,, Stir up the Athenian
    youth to merriments ;, Awake the pert and nimble spirit of mirth ;,
     Turn melancholy forth to funerals ;, The pale companion is not for
     our pomp ., "", Hippolyta , I woo'd thee with my sword ,, And won
    thy lo...
```