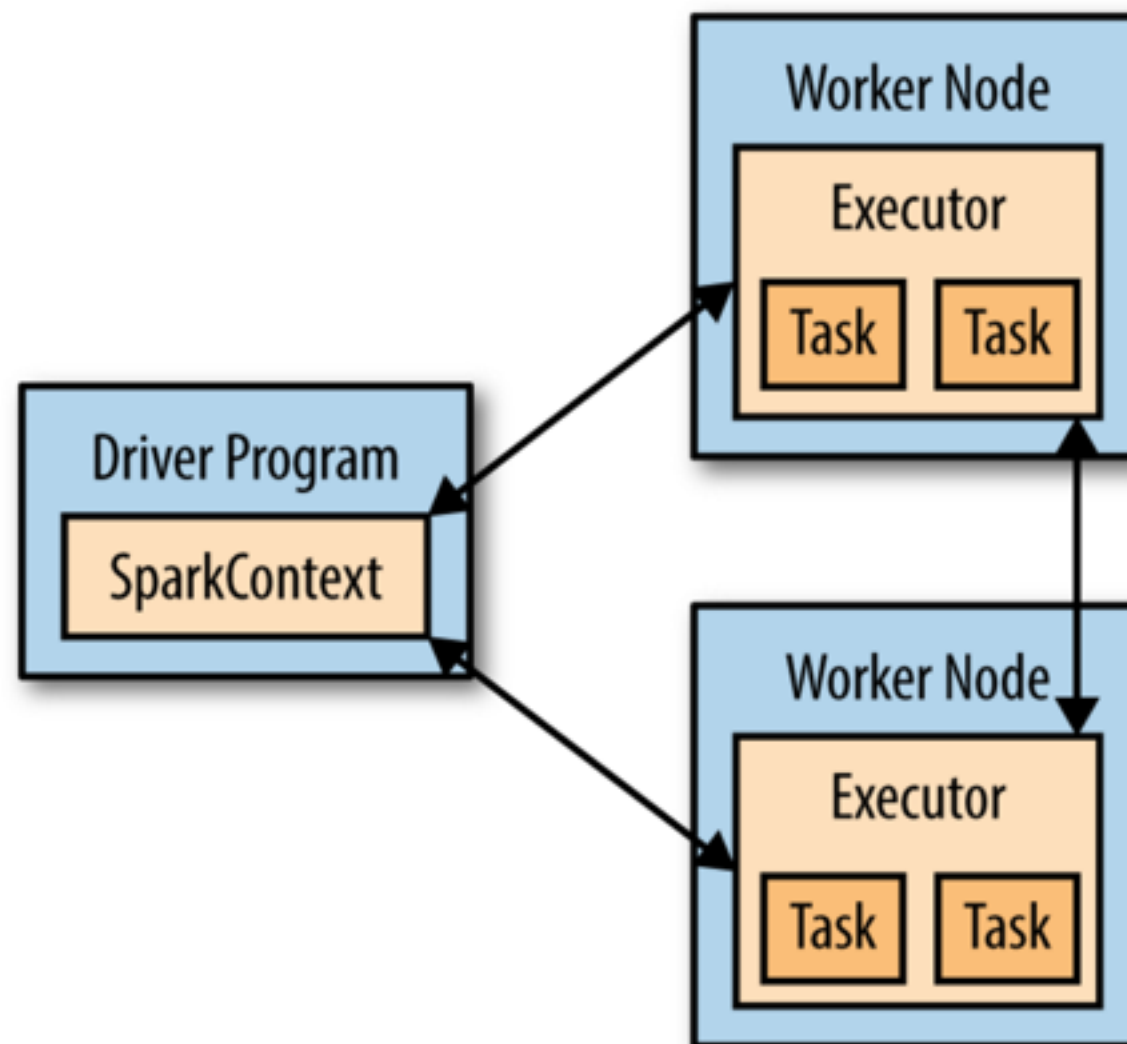


Spark




```

scala> pythonLines.first()
scala> val lines = sc.textFile("README.md")
lines: org.apache.spark.rdd.RDD[String] = README.md MapPartitionsRDD[1] at textFile at <console>:24

scala> lines.filter(_.contains("Python"))
res0: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[2] at filter at <console>:27

scala> val pythonLines = lines.filter(_.contains("Python"))
pythonLines: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[3] at filter at <console>:26

scala> pythonLines.count()
res1: Long = 3

scala> pythonLines.first()
res2: String = high-level APIs in Scala, Java, Python, and R, and an optimized engine that

scala>

```

```

scala> def hasPythonInLine(line: String) = line.contains ("Python")
hasPythonInLine: (line: String)Boolean

scala> val pythonLines = lines.filter(hasPythonInLine)
pythonLines: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[4] at filter at <console>:28

scala> pythonLines.count()
res3: Long = 3

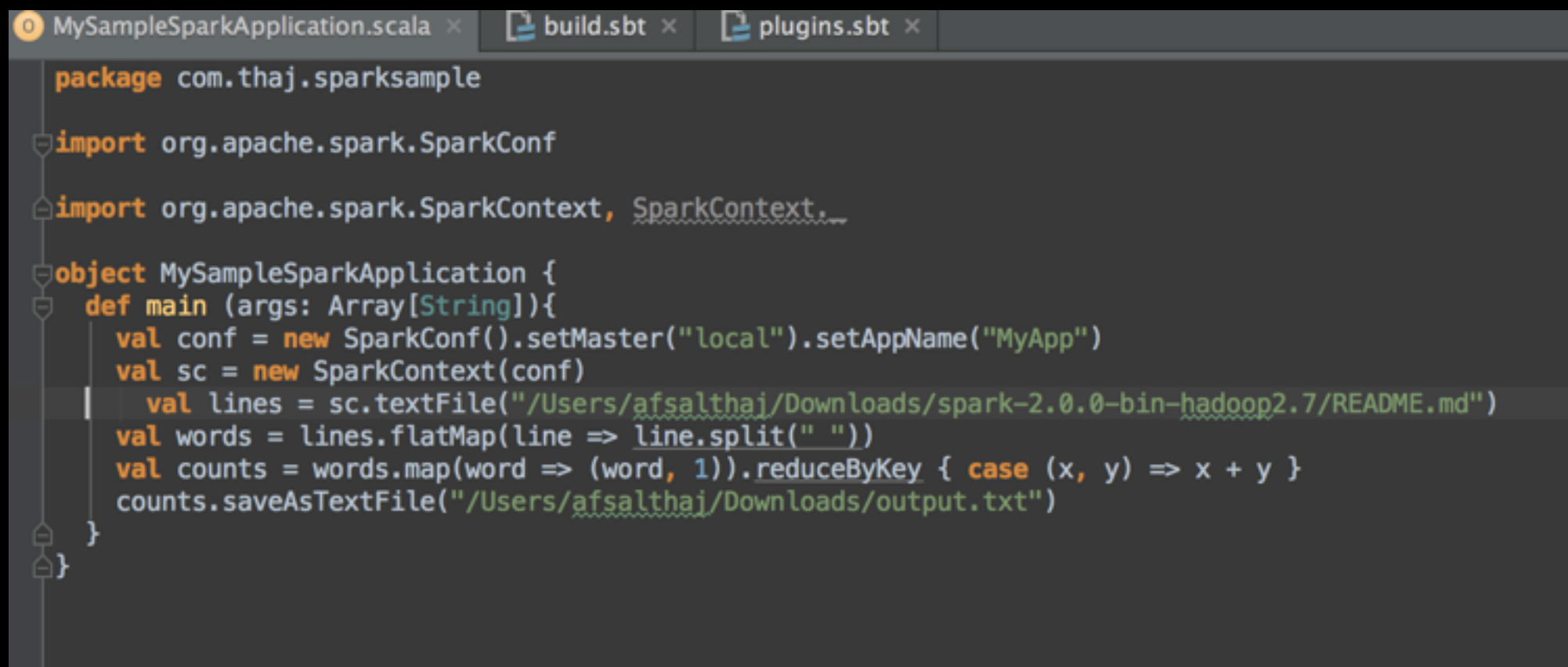
scala> pythonLines.first()
res4: String = high-level APIs in Scala, Java, Python, and R, and an optimized engine that

scala>

```

Spark Standalone Applications

Sample standalone app



The screenshot shows an IDE window with three tabs: 'MySampleSparkApplication.scala', 'build.sbt', and 'plugins.sbt'. The 'MySampleSparkApplication.scala' tab is active, displaying the following Scala code:

```
package com.thaj.sparksample

import org.apache.spark.SparkConf
import org.apache.spark.SparkContext, SparkContext._

object MySampleSparkApplication {
  def main (args: Array[String]){
    val conf = new SparkConf().setMaster("local").setAppName("MyApp")
    val sc = new SparkContext(conf)
    val lines = sc.textFile("/Users/afsalthaj/Downloads/spark-2.0.0-bin-hadoop2.7/README.md")
    val words = lines.flatMap(line => line.split(" "))
    val counts = words.map(word => (word, 1)).reduceByKey { case (x, y) => x + y }
    counts.saveAsTextFile("/Users/afsalthaj/Downloads/output.txt")
  }
}
```

RDD

the resilient distributed dataset (RDD)

Spark's core abstraction for working with data

An RDD is simply a distributed collection of elements

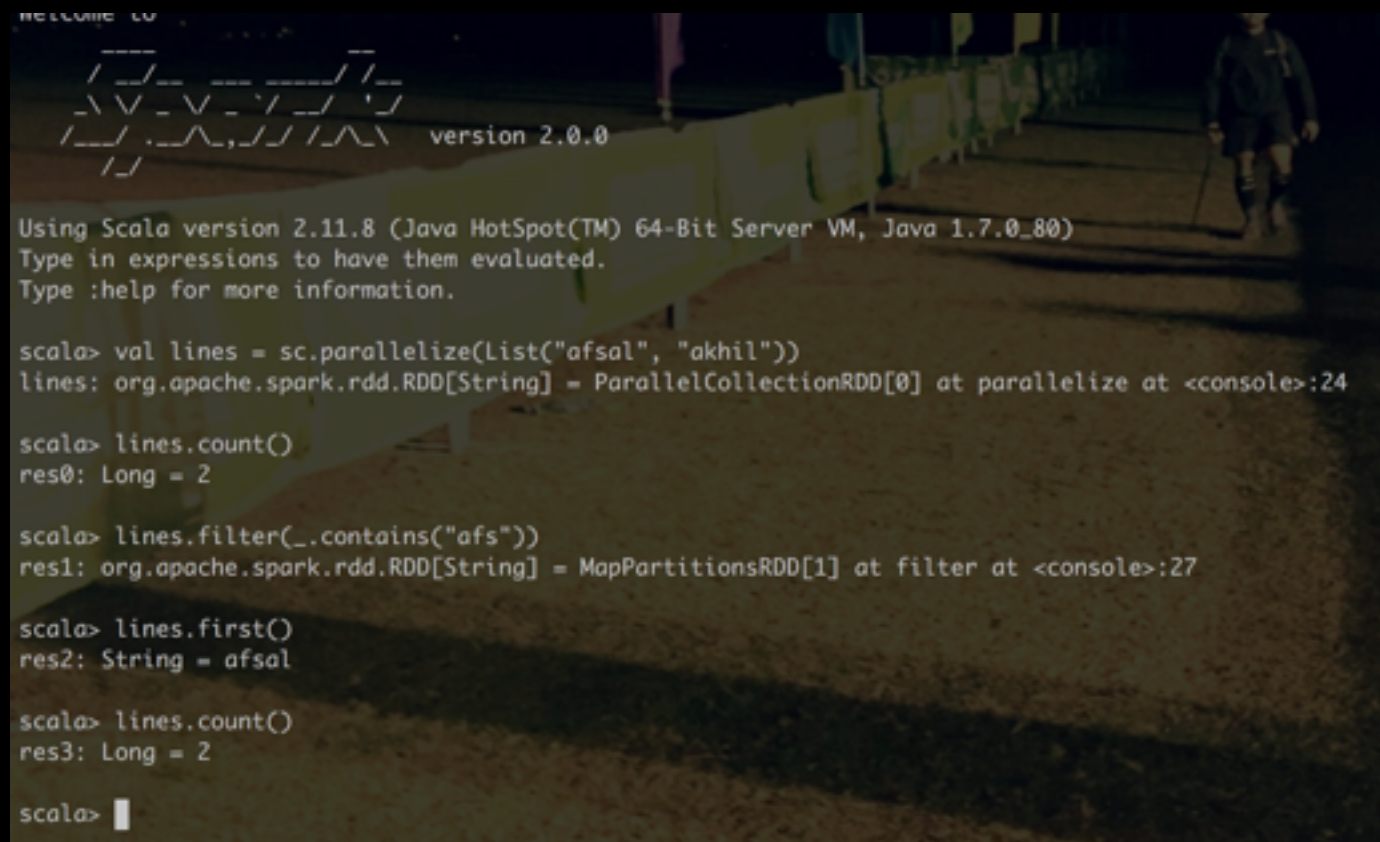
Once created, RDDs offer two types of operations: transformations and actions

RDD Operation

- Create some input RDDs from external data.
- Transform them to define new RDDs using transformations like `filter()`.
- Ask Spark to `persist()` any intermediate RDDs that will need to be reused.
- Launch actions such as `count()` and `first()` to kick off a parallel computation, which is then optimized and executed by Spark.

Creating RDD

- Spark provides two ways to create RDDs: loading an external dataset and parallelizing a collection in your driver program.
- `lines = sc.parallelize(["pandas", "i like pandas"])`

A screenshot of a Scala REPL (REPL) window. The background is a dark image of a person walking on a path. The text in the REPL shows the following sequence of commands and outputs:

```
welcome to ... version 2.0.0

Using Scala version 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_80)
Type in expressions to have them evaluated.
Type :help for more information.

scala> val lines = sc.parallelize(List("afsal", "akhil"))
lines: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[0] at parallelize at <console>:24

scala> lines.count()
res0: Long = 2

scala> lines.filter(_.contains("afs"))
res1: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[1] at filter at <console>:27

scala> lines.first()
res2: String = afsal

scala> lines.count()
res3: Long = 2

scala> █
```

More on RDD Operation

- Transformations return RDDs, whereas actions return some other data type.
- RDDs are computed lazily, only when you use them in an action
- Many transformations are element-wise; that is, they work on one element at a time; but this is not true for all transformations.

Immutable RDD

Please note, `union()` is a bit different than `filter()`, in that it operates on two RDDs instead of one. Transformations can actually operate on any number of input RDDs.

```
scala> val inputRDD = sc.textFile("log.txt")
inputRDD: org.apache.spark.rdd.RDD[String] = log.txt MapPartitionsRDD[3] at textFile at <console>:24

scala> val errorsRDD = inputRDD.filter(line => line.contains("error"))
errorsRDD: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[4] at filter at <console>:26

scala> val warningsRDD = inputRDD.filter(_.contains("warning"))
warningsRDD: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[5] at filter at <console>:26

scala> val badRDD = errorsRDD.union(warningsRDD)
badRDD: org.apache.spark.rdd.RDD[String] = UnionRDD[6] at union at <console>:30
```

More on Actions

- Operations that return a final value to the driver program or write data to an external storage system

```
scala> val inputRDD = sc.textFile("README.md")
inputRDD: org.apache.spark.rdd.RDD[String] = README.md MapPartitionsRDD[8] at textFile at <console>:24

scala> val errorsRDD = inputRDD.filter(line => line.contains("error"))
errorsRDD: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[9] at filter at <console>:26

scala> val warningsRDD = inputRDD.filter(_.contains("warning"))
warningsRDD: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[10] at filter at <console>:26

scala> val badRDD = errorsRDD.union(warningsRDD)
badRDD: org.apache.spark.rdd.RDD[String] = UnionRDD[11] at union at <console>:30

scala> badRDD.count()
res5: Long = 0

scala> badRDD.take(10).foreach(println)

scala>
```

- RDDs also have a `collect()` function to retrieve the entire RDD.
- `collect()` shouldn't be used on large datasets.
- In most cases RDDs can't just be `collect()`ed to the driver because they are too large. In these cases, it's common to write data out to a distributed storage system such as HDFS or Amazon S3
- You can save the contents of an RDD using the `saveAsTextFile()` action, `saveAsSequenceFile()`

- Lazy Evaluation: The core concept of SPARK: In Spark, there is no substantial benefit to writing a single complex map instead of chaining together many simple operations. Thus, users are free to organise their program into smaller, more manageable operations.

Common Transformations

Table 3-2. Basic RDD transformations on an RDD containing {1, 2, 3, 3}

Function name	Purpose	Example	Result
<code>map()</code>	Apply a function to each element in the RDD and return an RDD of the result.	<code>rdd.map(x => x + 1)</code>	{2, 3, 4, 4}
<code>flatMap()</code>	Apply a function to each element in the RDD and return an RDD of the contents of the iterators returned. Often used to extract words.	<code>rdd.flatMap(x => x.to(3))</code>	{1, 2, 3, 2, 3, 3, 3}
<code>filter()</code>	Return an RDD consisting of only elements that pass the condition passed to <code>filter()</code> .	<code>rdd.filter(x => x != 1)</code>	{2, 3, 3}
<code>distinct()</code>	Remove duplicates.	<code>rdd.distinct()</code>	{1, 2, 3}
<code>sample(withReplacement, fraction, [seed])</code>	Sample an RDD, with or without replacement.	<code>rdd.sample(false, 0.5)</code>	Nondeterministic

Table 3-3. Two-RDD transformations on RDDs containing {1, 2, 3} and {3, 4, 5}

Function name	Purpose	Example	Result
<code>union()</code>	Produce an RDD containing elements from both RDDs.	<code>rdd.union(other)</code>	{1, 2, 3, 3, 4, 5}
<code>intersection()</code>	RDD containing only elements found in both RDDs.	<code>rdd.intersection(other)</code>	{3}
<code>subtract()</code>	Remove the contents of one RDD (e.g., remove training data).	<code>rdd.subtract(other)</code>	{1, 2}
<code>cartesian()</code>	Cartesian product with the other RDD.	<code>rdd.cartesian(other)</code>	{(1, 3), (1, 4), ..., (3, 5)}

Common Actions

```
scala> list.aggregate(0)((acc, value) => (acc + value), (acc1, acc2) => acc1+acc2)
res3: Int = 6

scala> list.aggregate(0,0)((acc, value) => (acc._1 + 1, acc._2+value), (acc1, acc2) => (acc1._1+acc2._1, acc1._2+ acc1._2)
| )
res4: (Int, Int) = (3,0)

scala> list.aggregate(0,0)((acc, value) => (acc._1 + 1, acc._2+value), (acc1, acc2) => (acc1._1+acc2._1, acc1._2+ acc2._2)
| )
res5: (Int, Int) = (3,6)

scala> █
```

```
Using Scala version 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_80)
Type in expressions to have them evaluated.
Type :help for more information.

scala> val list = sc.parallelize(List(1,2,3))
list: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>:24

scala> list.reduce((a,b) => a+b)
res0: Int = 6

scala> list.reduce(_ + _)
res1: Int = 6
```

Some actions on RDDs return some or all of the data to our driver program in the form of a regular collection or value.

The simplest and most common operation that returns data to our driver program is `collect()`, which returns the entire RDD's contents. `collect()` is commonly used in unit tests where the entire contents of the RDD are expected to fit in memory, as that makes it easy to compare the value of our RDD with our expected result. `collect()` suffers from the restriction that all of your data must fit on a single machine, as it all needs to be copied to the driver.

`take(n)` returns `n` elements from the RDD and attempts to minimize the number of partitions it accesses, so it may represent a biased collection. It's important to note that these operations do not return the elements in the order you might expect.

Persistence

Table 3-6. Persistence levels from `org.apache.spark.storage.StorageLevel` and `pyspark.StorageLevel`; if desired we can replicate the data on two machines by adding `_2` to the end of the storage level

Level	Space used	CPU time	In memory	On disk	Comments
MEMORY_ONLY	High	Low	Y	N	
MEMORY_ONLY_SER	Low	High	Y	N	
MEMORY_AND_DISK	High	Medium	Some	Some	Spills to disk if there is too much data to fit in memory.
MEMORY_AND_DISK_SER	Low	High	Some	Some	Spills to disk if there is too much data to fit in memory. Stores serialized representation in memory.
DISK_ONLY	Low	High	N	Y	

```
res1.persist("MEMORY_ONLY")
^
scala> import org.apache.spark.storage.StorageLevel._
import org.apache.spark.storage.StorageLevel._

scala> res1.persist(MEMORY_ONLY)
res6: res1.type = MapPartitionsRDD[2] at map at <console>:27

scala> res1.persist(MEMORY_ONLY_SER)
java.lang.UnsupportedOperationException: Cannot change storage level of an RDD after it was already assigned a level
    at org.apache.spark.rdd.RDD.persist(RDD.scala:169)
    at org.apache.spark.rdd.RDD.persist(RDD.scala:194)
    ... 50 elided

scala> 
```

Spark Key-Value Pairs

Spark provides special operations on RDDs containing key/value pairs. These RDDs are called pair RDDs. Pair RDDs are a useful building block in many programs, as they expose operations that allow you to act on each key in parallel or regroup data across the network. For example, pair RDDs have a `reduceByKey()` method that can aggregate data separately for each key, and a `join()` method that can merge two RDDs together by grouping elements with the same key. It is common to extract fields from an RDD (representing, for instance, an event time, customer ID, or other identifier) and use those fields as keys in pair RDD operations.

Key-Value Operations

```
scala> val list = sc.parallelize(List((1, 2), (3, 4), (3, 6)))
      | )
list: org.apache.spark.rdd.RDD[(Int, Int)] = ParallelCollectionRDD[5] at parallelize at <console>:24

scala> list.reduceByKey(_ + _)
res8: org.apache.spark.rdd.RDD[(Int, Int)] = ShuffledRDD[6] at reduceByKey at <console>:27

scala> list.collect
res9: Array[(Int, Int)] = Array((1,2), (3,4), (3,6))

scala> res8.collect
res10: Array[(Int, Int)] = Array((1,2), (3,10))

scala> █
```

Key-Value Operations

```
scala> list.groupByKey()
res3: org.apache.spark.rdd.RDD[(Int, Iterable[Int])] = ShuffledRDD[2] at groupByKey at <console>:27
```

```
scala> res3.collect
res4: Array[(Int, Iterable[Int])] = Array((1,CompactBuffer(2)), (3,CompactBuffer(4, 6)))
```

```
scala> res3.collect.toList
res5: List[(Int, Iterable[Int])] = List((1,CompactBuffer(2)), (3,CompactBuffer(4, 6)))
```

```
scala>
```

```
scala> list.mapValues(_ + 1)
res6: org.apache.spark.rdd.RDD[(Int, Int)] = MapPartitionsRDD[3] at mapValues at <console>:27
```

```
scala> res6.collect
res7: Array[(Int, Int)] = Array((1,3), (3,5), (3,7))
```

```
scala>
```

```
scala> list.flatMapValues(x => (x to 5))
res8: org.apache.spark.rdd.RDD[(Int, Int)] = MapPartitionsRDD[4] at flatMapValues at <console>:27
```

```
scala> res8.collect
res9: Array[(Int, Int)] = Array((1,2), (1,3), (1,4), (1,5), (3,4), (3,5))
```

Key-Value Operations

```
scala> list.keys  
res10: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[5] at keys at <console>:27
```

```
scala> res10.collect  
res11: Array[Int] = Array(1, 3, 3)
```

```
scala> list.values  
res12: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[6] at values at <console>:27
```

```
scala> res12.collect  
res13: Array[Int] = Array(2, 4, 6)
```

```
scala> val list = sc.parallelize(List((3,4), (1,2), (3,6))  
    | )  
list: org.apache.spark.rdd.RDD[(Int, Int)] = ParallelCollectionRDD[9] at parallelize at <console>:24
```

```
scala> list.sortByKey  
<console>:27: error: missing argument list for method sortByKey in class OrderedRDDFunctions  
Unapplied methods are only converted to functions when a function type is expected.  
You can make this conversion explicit by writing `sortByKey _` or `sortByKey(_,_)` instead of `sortByKey`.  
    list.sortByKey  
           ^
```

```
scala> list.sortByKey()  
res17: org.apache.spark.rdd.RDD[(Int, Int)] = ShuffledRDD[12] at sortByKey at <console>:27
```

```
scala> res17.collect  
res18: Array[(Int, Int)] = Array((1,2), (3,4), (3,6))
```

```
scala>
```


Key Value Operations

```
scala> val rdd = sc.parallelize(List((1,2),(3,4),(3,6))
    | )
rdd: org.apache.spark.rdd.RDD[(Int, Int)] = ParallelCollectionRDD[13] at parallelize at <console>:24
```

```
scala> val other = sc.parallelize(List((3,9)))
other: org.apache.spark.rdd.RDD[(Int, Int)] = ParallelCollectionRDD[14] at parallelize at <console>:24
```

```
scala> rdd.subtract
subtract    subtractByKey
```

```
scala> rdd.subtractByKey(other)
res19: org.apache.spark.rdd.RDD[(Int, Int)] = SubtractedRDD[15] at subtractByKey at <console>:29
```

```
scala> res19.collect
res20: Array[(Int, Int)] = Array((1,2))
```

```
scala>
```

```
scala> rdd.join(other)
res21: org.apache.spark.rdd.RDD[(Int, (Int, Int))] = MapPartitionsRDD[18] at join at <console>:29
```

```
scala> res21.collect
res22: Array[(Int, (Int, Int))] = Array((3,(4,9)), (3,(6,9)))
```

```
scala>
```

Key Value Operations

```
scala> rdd.rightOuterJoin(other)
res23: org.apache.spark.rdd.RDD[(Int, (Option[Int], Int))] = MapPartitionsRDD[21] at rightOuterJoin at <console>:29
```

```
scala> res23.collect
res24: Array[(Int, (Option[Int], Int))] = Array((3,(Some(4),9)), (3,(Some(6),9)))
```

```
scala> res23.collect.toList
res25: List[(Int, (Option[Int], Int))] = List((3,(Some(4),9)), (3,(Some(6),9)))
```

```
scala>
```

```
scala> rdd.leftOuterJoin(other)
res26: org.apache.spark.rdd.RDD[(Int, (Int, Option[Int]))] = MapPartitionsRDD[24] at leftOuterJoin at <console>:29
```

```
scala> res26.collect
res27: Array[(Int, (Int, Option[Int]))] = Array((1,(2,None)), (3,(4,Some(9))), (3,(6,Some(9))))
```

```
scala>
```

```
scala> rdd.cogroup(other)
res28: org.apache.spark.rdd.RDD[(Int, (Iterable[Int], Iterable[Int]))] = MapPartitionsRDD[26] at cogroup at <console>:29
```

```
scala> res28.collect
res29: Array[(Int, (Iterable[Int], Iterable[Int]))] = Array((1,(CompactBuffer(2),CompactBuffer()), (3,(CompactBuffer(4,6),CompactBuffer(9))))
```

```
scala> res29.toList
res30: List[(Int, (Iterable[Int], Iterable[Int]))] = List((1,(CompactBuffer(2),CompactBuffer()), (3,(CompactBuffer(4,6),CompactBuffer(9))))
```

```
scala>
```

Key Value Actions

Actions Available on Pair RDDs

As with the transformations, all of the traditional actions available on the base RDD are also available on pair RDDs. Some additional actions are available on pair RDDs to take advantage of the key/value nature of the data; these are listed in [Table 4-3](#).

Table 4-3. Actions on pair RDDs (example $\{(1, 2), (3, 4), (3, 6)\}$)

Function	Description	Example	Result
<code>countByKey()</code>	Count the number of elements for each key.	<code>rdd.countByKey()</code>	<code>{{(1, 1), (3, 2)}}</code>
<code>collectAsMap()</code>	Collect the result as a map to provide easy lookup.	<code>rdd.collectAsMap()</code>	<code>Map[(1, 2), (3, 4), (3, 6)]</code>
<code>lookup(key)</code>	Return all values associated with the provided key.	<code>rdd.lookup(3)</code>	<code>[4, 6]</code>

There are also multiple other actions on pair RDDs that save the RDD, which we will describe in [Chapter 5](#).

<http://spark.apache.org/docs/latest/sql-programming-guide.html>