Chapter 3. Programming with RDDs

This chapter introduces Spark's core abstraction for working with data, the resilient distributed dataset (RDD). An RDD is simply a distributed collection of elements. In Spark all work is expressed as either creating new RDDs, transforming existing RDDs, or calling operations on RDDs to compute a result. Under the hood, Spark automatically distributes the data contained in RDDs across your cluster and parallelizes the operations you perform on them.

Both data scientists and engineers should read this chapter, as RDDs are the core concept in Spark. We highly recommend that you try some of these examples in an interactive shell (see "Introduction to Spark's Python and Scala Shells"). In addition, all code in this chapter is available in the book's GitHub repository.

RDD Basics

An RDD in Spark is simply an immutable distributed collection of objects. Each RDD is split into multiple *partitions*, which may be computed on different nodes of the cluster. RDDs can contain any type of Python, Java, or Scala objects, including user-defined classes.

Users create RDDs in two ways: by loading an external dataset, or by distributing a collection of objects (e.g., a list or set) in their driver program. We have already seen loading a text file as an RDD of strings using `SparkContext.textFile()`, as shown in Example 3-1.

Example 3-1. Creating an RDD of strings with textFile() in Python

```
>>> lines = sc.textFile("README.md")
```

Once created, RDDs offer two types of operations: *transformations* and *actions*. *Transformations* construct a new RDD from a previous one. For example, one common transformation is filtering data that matches a predicate. In our text file example, we can use this to create a new RDD holding just the strings that contain the word *Python*, as shown in Example 3-2.

Example 3-2. Calling the filter() transformation

```
>>> pythonLines = lines.filter(lambda line: "Python" in line)
```

Actions, on the other hand, compute a result based on an RDD, and either return it to the driver program or save it to an external storage system (e.g., HDFS). One example of an action we called earlier is `first()`, which returns the first element in an RDD and is demonstrated in Example 3-3.

Example 3-3. Calling the first() action

```
>>> pythonLines.first()
u'## Interactive Python Shell'
```

Transformations and actions are different because of the way Spark computes RDDs. Although you can define new RDDs any time, Spark computes them only in a *lazy* fashion—that is, the first time they are used in an action. This approach might seem unusual at first, but makes a lot of sense when you are working with Big Data. For instance, consider Example 3-2 and Example 3-3, where we defined a text file and then filtered the lines that include *Python*. If Spark were to load and store all the lines in the file as soon as we wrote `lines = sc.textFile(...)`, it would waste a lot of storage space, given that we then immediately filter out many lines. Instead, once Spark sees the whole chain of transformations, it can compute just the data needed for its result. In fact, for the `first()` action, Spark scans the file only until it finds the first matching line; it doesn't even read the whole file.

Finally, Spark's RDDs are by default recomputed each time you run an action on them. If you would like to reuse an RDD in multiple actions, you can ask Spark to *persist* it using `RDD.persist()`. We can ask Spark to persist our data in a number of different places, which will be covered in Table 3-6. After computing it the first time, Spark will store the RDD contents in memory (partitioned across the machines in your cluster), and reuse them in future actions. Persisting RDDs on disk instead of memory is also possible. The behavior of not persisting by default may again seem unusual, but it makes a lot of sense for big datasets: if you will not reuse the RDD, there's no reason to waste storage space when Spark could instead stream through the data once and just compute the result.

In practice, you will often use `persist()` to load a subset of your data into memory and query it repeatedly. For example, if we knew that we wanted to compute multiple results about the README lines that contain *Python*, we could write the script shown in Example 3-4.

Example 3-4. Persisting an RDD in memory

storage level.

In the rest of this chapter, we'll go through each of these steps in detail, and cover some of the most common RDD operations in Spark.

Creating RDDs

Spark provides two ways to create RDDs: loading an external dataset and parallelizing a collection in your driver program.

The simplest way to create RDDs is to take an existing collection in your program and pass it to SparkContext's `parallelize()` method, as shown in Examples 3-5 through 3-7. This approach is very useful when you are learning Spark, since you can quickly create your own RDDs in the shell and perform operations on them. Keep in mind, however, that outside of prototyping and testing, this is not widely used since it requires that you have your entire dataset in memory on one machine.

Example 3-5. parallelize() method in Python

```
lines = sc.parallelize(["pandas", "i like pandas"])
```

Example 3-6. parallelize() method in Scala

```
val lines = sc.parallelize(List("pandas", "i like pandas"))
```

Example 3-7. parallelize() method in Java

```
JavaRDD<String> lines = sc.parallelize(Arrays.asList("pandas", "
```

A more common way to create RDDs is to load data from external storage. Loading external datasets is covered in detail in Chapter 5. However, we already saw one method that loads a text file as an RDD of strings, `SparkContext.textFile()`, which is shown in Examples 3-8 through 3-10.

Example 3-8. textFile() method in Python

```
lines = sc.textFile("/path/to/README.md")
```

Example 3-9. textFile() method in Scala

```
val lines = sc.textFile("/path/to/README.md")
```

Example 3-10. textFile() method in Java

```
JavaRDD<String> lines = sc.textFile("/path/to/README.md");
```

RDD Operations

As we've discussed, RDDs support two types of operations: *transformations* and *actions*. Transformations are operations on RDDs that return a new RDD, such as `map()` and `filter()`. Actions are operations that return a result to the driver program or write it to storage, and kick off a computation, such as `count()` and `first()`. Spark treats transformations and actions very differently, so understanding which type of operation you are performing will be important. If you are ever confused whether a given function is a transformation or an action, you can look at its return type: transformations return RDDs, whereas actions return some other data type.

Transformations

Transformations are operations on RDDs that return a new RDD. As discussed in "Lazy Evaluation", transformed 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.

As an example, suppose that we have a logfile, *log.txt*, with a number of messages, and we want to select only the error messages. We can use the `filter()` transformation seen before. This time, though, we'll show a filter in all three of Spark's language APIs (Examples 3-11 through 3-13).

Example 3-11. filter() transformation in Python

```
inputRDD = sc.textFile("log.txt")
errorsRDD = inputRDD.filter(lambda x: "error" in x)
```

Example 3-12. filter() transformation in Scala

```
val inputRDD = sc.textFile("log.txt")
val errorsRDD = inputRDD.filter(line => line.contains("error"))
```

`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.

> **TIP**
>
> A better way to accomplish the same result as in Example 3-14 would be to simply filter the `inputRDD` once, looking for either *error* or *warning*.

Finally, as you derive new RDDs from each other using transformations, Spark keeps track of the set of dependencies between different RDDs, called the *lineage graph*. It uses this information to compute each RDD on demand and to recover lost data if part of a persistent RDD is lost. Figure 3-1 shows a lineage graph for Example 3-14.

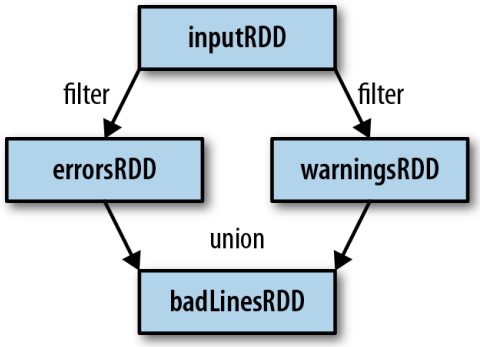

Figure 3-1. RDD lineage graph created during log analysis

Actions

We've seen how to create RDDs from each other with transformations, but at some point, we'll want to actually *do something* with our dataset. Actions are the second type of RDD operation. They are the operations that return a final value to the driver program or write data to an external storage system. Actions force the evaluation of the transformations required for the RDD they were called on, since they need to actually produce output.

Continuing the log example from the previous section, we might want to print out some information about the `badLinesRDD`. To do that, we'll use two actions, `count()`, which returns the count as a number, and `take()`, which collects a number of elements from the RDD, as shown in Examples 3-15 through 3-17.

Example 3-15. Python error count using actions

```
print "Input had " + badLinesRDD.count() + " concerning lines"
print "Here are 10 examples:"
for line in badLinesRDD.take(10):
    print line
```

Example 3-16. Scala error count using actions

```
println("Input had " + badLinesRDD.count() + " concerning lines"
println("Here are 10 examples:")
badLinesRDD.take(10).foreach(println)
```

Example 3-17. Java error count using actions

```
System.out.println("Input had " + badLinesRDD.count() + " concer
System.out.println("Here are 10 examples:")
for (String line: badLinesRDD.take(10)) {
  System.out.println(line);
}
```

In this example, we used `take()` to retrieve a small number of elements in the RDD at the driver program. We then iterate over them locally to print out information at the driver. RDDs also have a `collect()` function to retrieve the entire RDD. This can be useful if your program filters RDDs down to a very small size and you'd like to deal with it locally. Keep in mind that your entire dataset must fit in memory on a single machine to use `collect()` on it, so `collect()` shouldn't be used on large datasets.

formations. Loading data into an RDD is lazily evaluated in the same way transformations are. So, when we call `sc.textFile()`, the data is not loaded until it is necessary. As with transformations, the operation (in this case, reading the data) can occur multiple times.

> **TIP**
>
> Although transformations are lazy, you can force Spark to execute them at any time by running an action, such as `count()`. This is an easy way to test out just part of your program.

Spark uses lazy evaluation to reduce the number of passes it has to take over our data by grouping operations together. In systems like Hadoop MapReduce, developers often have to spend a lot of time considering how to group together operations to minimize the number of MapReduce passes. 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 organize their program into smaller, more manageable operations.

Passing Functions to Spark

Most of Spark's transformations, and some of its actions, depend on passing in functions that are used by Spark to compute data. Each of the core languages has a slightly different mechanism for passing functions to Spark.

Python

In Python, we have three options for passing functions into Spark. For shorter functions, we can pass in lambda expressions, as we did in Example 3-2, and as Example 3-18 demonstrates. Alternatively, we can pass in top-level functions, or locally defined functions.

Example 3-18. Passing functions in Python

```
word = rdd.filter(lambda s: "error" in s)
```

```
def containsError(s):
    return "error" in s
word = rdd.filter(containsError)
```

One issue to watch out for when passing functions is inadvertently serializing the object containing the function. When you pass a function that is the member of an object, or contains references to fields in an object (e.g., `self.field`), Spark sends the *entire* object to worker nodes, which can be much larger than the bit of information you need (see Example 3-19). Sometimes this can also cause your program to fail, if your class contains objects that Python can't figure out how to pickle.

Example 3-19. Passing a function with field references (don't do this!)

```
class SearchFunctions(object):
  def __init__(self, query):
      self.query = query
  def isMatch(self, s):
      return self.query in s
  def getMatchesFunctionReference(self, rdd):
      # Problem: references all of "self" in "self.isMatch"
      return rdd.filter(self.isMatch)
  def getMatchesMemberReference(self, rdd):
      # Problem: references all of "self" in "self.query"
      return rdd.filter(lambda x: self.query in x)
```

Instead, just extract the fields you need from your object into a local variable and pass that in, like we do in Example 3-20.

Example 3-20. Python function passing without field references

```
class WordFunctions(object):
  ...
  def getMatchesNoReference(self, rdd):
      # Safe: extract only the field we need into a local variab
      query = self.query
      return rdd.filter(lambda x: query in x)
```

Scala

In Scala, we can pass in functions defined inline, references to methods, or static functions as we do for Scala's other functional APIs. Some other considerations come into play, though—namely that the function we pass and the data referenced in

```
}
```

If `NotSerializableException` occurs in Scala, a reference to a method or field in a nonserializable class is usually the problem. Note that passing in local serializable variables or functions that are members of a top-level object is always safe.

Java

In Java, functions are specified as objects that implement one of Spark's function interfaces from the `org.apache.spark.api.java.function` package. There are a number of different interfaces based on the return type of the function. We show the most basic function interfaces in Table 3-1, and cover a number of other function interfaces for when we need to return special types of data, like key/value data, in "Java".

Table 3-1. Standard Java function interfaces

Function name	Method to implement	Usage
`Function<T, R>`	`R call(T)`	Take in one input and return one output, for use with operations like `map()` and `filter()`.
`Function2<T1, T2, R>`	`R call(T1, T2)`	Take in two inputs and return one output, for use with operations like `aggregate()` or `fold()`.
`FlatMapFunction<T, R>`	`Iterable<R> call(T)`	Take in one input and return zero or more outputs, for use with operations like `flatMap()`.

We can either define our function classes inline as anonymous inner classes (Example 3-22), or create a named class (Example 3-23).

Example 3-22. Java function passing with anonymous inner class

```
RDD<String> errors = lines.filter(new Function<String, Boolean>
  public Boolean call(String x) { return x.contains("error");
});
```

Example 3-23. Java function passing with named class

```
class ContainsError implements Function<String, Boolean>()
  public Boolean call(String x) { return x.contains("error");
}

RDD<String> errors = lines.filter(new ContainsError());
```

The style to choose is a personal preference, but we find that top-level named functions are often cleaner for organizing large programs. One other benefit of top-level functions is that you can give them constructor parameters, as shown in Example 3-24.

Example 3-24. Java function class with parameters

```
class Contains implements Function<String, Boolean>() {
  private String query;
  public Contains(String query) { this.query = query; }
  public Boolean call(String x) { return x.contains(query); }
}

RDD<String> errors = lines.filter(new Contains("error"));
```

In Java 8, you can also use lambda expressions to concisely implement the function interfaces. Since Java 8 is still relatively new as of this writing, our examples use the more verbose syntax for defining classes in previous versions of Java. However, with lambda expressions, our search example would look like Example 3-25.

Example 3-25. Java function passing with lambda expression in Java 8

aggregating data by key on RDDs of key/value pairs. We cover converting between RDD types and these special operations in later sections.

Basic RDDs

We will begin by describing what transformations and actions we can perform on all RDDs regardless of the data.

ELEMENT-WISE TRANSFORMATIONS

The two most common transformations you will likely be using are `map()` and `filter()` (see Figure 3-2). The `map()` transformation takes in a function and applies it to each element in the RDD with the result of the function being the new value of each element in the resulting RDD. The `filter()` transformation takes in a function and returns an RDD that only has elements that pass the `filter()` function.

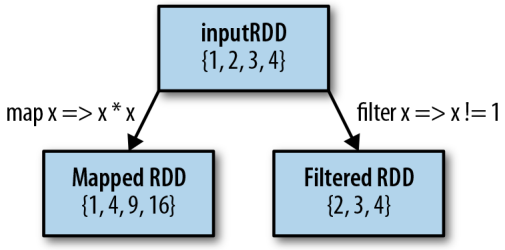

Figure 3-2. Mapped and filtered RDD from an input RDD

We can use `map()` to do any number of things, from fetching the website associated with each URL in our collection to just squaring the numbers. It is useful to note that `map()`'s return type does not have to be the same as its input type, so if we had an RDD `String` and our `map()` function were to parse the strings and return a `Double`, our input RDD type would be `RDD[String]` and the resulting RDD type would be `RDD[Double]`.

Let's look at a basic example of `map()` that squares all of the numbers in an RDD (Examples 3-26 through 3-28).

Example 3-26. Python squaring the values in an RDD

```
nums = sc.parallelize([1, 2, 3, 4])
squared = nums.map(lambda x: x * x).collect()
for num in squared:
    print "%i " % (num)
```

Example 3-27. Scala squaring the values in an RDD

```
val input = sc.parallelize(List(1, 2, 3, 4))
val result = input.map(x => x * x)
println(result.collect().mkString(","))
```

Example 3-28. Java squaring the values in an RDD

```
JavaRDD<Integer> rdd = sc.parallelize(Arrays.asList(1, 2, 3, 4))
JavaRDD<Integer> result = rdd.map(new Function<Integer, Integer
  public Integer call(Integer x) { return x*x; }
});
System.out.println(StringUtils.join(result.collect(), ","));
```

Sometimes we want to produce multiple output elements for each input element. The operation to do this is called `flatMap()`. As with `map()`, the function we provide to `flatMap()` is called individually for each element in our input RDD. Instead of returning a single element, we return an iterator with our return values. Rather than producing an RDD of iterators, we get back an RDD that consists of the elements from all of the iterators. A simple usage of `flatMap()` is splitting up an input string into words, as shown in Examples 3-29 through 3-31.

Example 3-29. flatMap() in Python, splitting lines into words

```
lines = sc.parallelize(["hello world", "hi"])
words = lines.flatMap(lambda line: line.split(" "))
words.first()  # returns "hello"
```

Example 3-30. flatMap() in Scala, splitting lines into multiple words

```
val lines = sc.parallelize(List("hello world", "hi"))
val words = lines.flatMap(line => line.split(" "))
words.first()  // returns "hello"
```

Figure 3-3. Difference between flatMap() and map() on an RDD

PSEUDO SET OPERATIONS

RDDs support many of the operations of mathematical sets, such as union and intersection, even when the RDDs themselves are not properly sets. Four operations are shown in Figure 3-4. It's important to note that all of these operations require that the RDDs being operated on are of the same type.

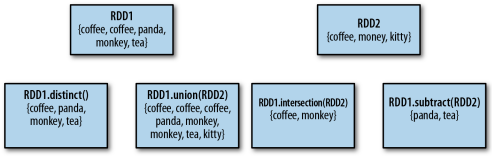

Figure 3-4. Some simple set operations

The set property most frequently missing from our RDDs is the uniqueness of elements, as we often have duplicates. If we want only unique elements we can use the `RDD.distinct()` transformation to produce a new RDD with only distinct items. Note that `distinct()` is expensive, however, as it requires shuffling all the data over the network to ensure that we receive only one copy of each element. Shuffling, and how to avoid it, is discussed in more detail in Chapter 4.

The simplest set operation is `union(other)`, which gives back an RDD consisting of the data from both sources. This can be useful in a number of use cases, such as processing logfiles from many sources. Unlike the mathematical `union()`, if there are duplicates in the input RDDs, the result of Spark's `union()` will contain duplicates (which we can fix if desired with `distinct()`).

Spark also provides an `intersection(other)` method, which returns only elements in both RDDs. `intersection()` also removes all duplicates (including duplicates from a single RDD) while running. While `intersection()` and `union()` are two similar concepts, the performance of `intersection()` is much worse since it requires a shuffle over the network to identify common elements.

Sometimes we need to remove some data from consideration. The `subtract(other)` function takes in another RDD and returns an RDD that has only values present in the first RDD and not the second RDD. Like `intersection()`, it performs a shuffle.

We can also compute a Cartesian product between two RDDs, as shown in Figure 3-5. The `cartesian(other)` transformation returns all possible pairs of `(a, b)` where `a` is in the source RDD and `b` is in the other RDD. The Cartesian product can be useful when we wish to consider the similarity between all possible pairs, such as computing every user's expected interest in each offer. We can also take the Cartesian product of an RDD with itself, which can be useful for tasks like user similarity. Be warned, however, that the Cartesian product is *very expensive* for large RDDs.

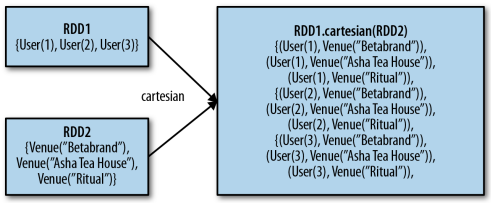

Figure 3-5. Cartesian product between two RDDs

Tables 3-2 and 3-3 summarize these and other common RDD transformations.

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

Function name	Purpose	Example	Res
`map()`	Apply a function to each element in the RDD and return an RDD	`rdd.map(x => x + 1)`	{2, 4,

`filter()`	Return an RDD consisting of only elements that pass the condition passed to `filter()`.	`rdd.filter(x => x != 1)`	`{2, 3}`
`distinct()`	Remove duplicates.	`rdd.distinct()`	`{1, 3}`
`sample(withReplacement, fraction, [seed])`	Sample an RDD, with or without replacement.	`rdd.sample(false, 0.5)`	Non-deterministic

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

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

ACTIONS

The most common action on basic RDDs you will likely use is `reduce()`, which takes a function that operates on two elements of the type in your RDD and returns a new element of the same type. A simple example of such a function is +, which we can use to sum our RDD. With `reduce()`, we can easily sum the elements of our RDD, count the number of elements, and perform other types of aggregations (see Examples 3-32 through 3-34).

Example 3-32. reduce() in Python

```
sum = rdd.reduce(lambda x, y: x + y)
```

you should not modify the second parameter.

Both `fold()` and `reduce()` require that the return type of our result be the same type as that of the elements in the RDD we are operating over. This works well for operations like `sum`, but sometimes we want to return a different type. For example, when computing a running average, we need to keep track of both the count so far and the number of elements, which requires us to return a pair. We could work around this by first using `map()` where we transform every element into the element and the number 1, which is the type we want to return, so that the `reduce()` function can work on pairs.

The `aggregate()` function frees us from the constraint of having the return be the same type as the RDD we are working on. With `aggregate()`, like `fold()`, we supply an initial zero value of the type we want to return. We then supply a function to combine the elements from our RDD with the accumulator. Finally, we need to supply a second function to merge two accumulators, given that each node accumulates its own results locally.

We can use `aggregate()` to compute the average of an RDD, avoiding a `map()` before the `fold()`, as shown in Examples 3-35 through 3-37.

Example 3-35. aggregate() in Python

```
sumCount = nums.aggregate((0, 0),
                          (lambda acc, value: (acc[0] + value, acc[1] + 1)
                          (lambda acc1, acc2: (acc1[0] + acc2[0], acc1[1]
return sumCount[0] / float(sumCount[1])
```

Example 3-36. aggregate() in Scala

```
val result = input.aggregate((0, 0))(
                          (acc, value) => (acc._1 + value, acc._2 + 1),
                          (acc1, acc2) => (acc1._1 + acc2._1, acc1._2 + acc
val avg = result._1 / result._2.toDouble
```

Example 3-37. aggregate() in Java

```
class AvgCount implements Serializable {
  public AvgCount(int total, int num) {
    this.total = total;
    this.num = num;
  }
  public int total;
  public int num;
  public double avg() {
    return total / (double) num;
  }
}
Function2<AvgCount, Integer, AvgCount> addAndCount =
  new Function2<AvgCount, Integer, AvgCount>() {
    public AvgCount call(AvgCount a, Integer x) {
      a.total += x;
      a.num += 1;
      return a;
  }
};
Function2<AvgCount, AvgCount, AvgCount> combine =
  new Function2<AvgCount, AvgCount, AvgCount>() {
  public AvgCount call(AvgCount a, AvgCount b) {
    a.total += b.total;
    a.num += b.num;
    return a;
  }
};
AvgCount initial = new AvgCount(0, 0);
AvgCount result = rdd.aggregate(initial, addAndCount, combine);
System.out.println(result.avg());
```

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.

These operations are useful for unit tests and quick debugging, but may introduce

Function name	Purpose	Example
`collect()`	Return all elements from the RDD.	`rdd.collect()`
`count()`	Number of elements in the RDD.	`rdd.count()`
`countByValue()`	Number of times each element occurs in the RDD.	`rdd.countByValue()`
`take(num)`	Return `num` elements from the RDD.	`rdd.take(2)`
`top(num)`	Return the top `num` elements the RDD.	`rdd.top(2)`
`takeOrdered(num)(ordering)`	Return `num` elements based on provided ordering.	`rdd.takeOrdered(2)(my`
`takeSample(withReplacement, num, [seed])`	Return `num` elements at random.	`rdd.takeSample(false,`
`reduce(func)`	Combine the elements of the RDD together in parallel (e.g., `sum`).	`rdd.reduce((x, y) =>`
`fold(zero)(func)`	Same as `reduce()` but with the provided zero value.	`rdd.fold(0)((x, y) =>`
`aggregate(zeroValue)(seqOp, combOp)`	Similar to `reduce()` but used to return a different	`rdd.aggregate((0, 0))` `=> (x._1 + y, x._2 + 1` `=> (x._1 + y._1, x._2`

SCALA

In Scala the conversion to RDDs with special functions (e.g., to expose numeric functions on an `RDD[Double]`) is handled automatically using implicit conversions. As mentioned in "Initializing a SparkContext", we need to add `import org.apache.spark.SparkContext._` for these conversions to work. You can see the implicit conversions listed in the `SparkContext` object's ScalaDoc (http://bit.ly/1Bc4fNt). These implicits turn an RDD into various wrapper classes, such as `DoubleRDDFunctions` (for RDDs of numeric data) and `PairRDDFunctions` (for key/value pairs), to expose additional functions such as `mean()` and `variance()`.

Implicits, while quite powerful, can sometimes be confusing. If you call a function like `mean()` on an RDD, you might look at the Scaladocs for the RDD class (http://bit.ly/1KiC7uO) and notice there is no `mean()` function. The call manages to succeed because of implicit conversions between `RDD[Double]` and `DoubleRDDFunctions`. When searching for functions on your RDD in Scaladoc, make sure to look at functions that are available in these wrapper classes.

JAVA

In Java the conversion between the specialized types of RDDs is a bit more explicit. In particular, there are special classes called `JavaDoubleRDD` and `JavaPairRDD` for RDDs of these types, with extra methods for these types of data. This has the benefit of giving you a greater understanding of what exactly is going on, but can be a bit more cumbersome.

To construct RDDs of these special types, instead of always using the `Function` class we will need to use specialized versions. If we want to create a `DoubleRDD` from an RDD of type `T`, rather than using `Function<T, Double>` we use `DoubleFunction<T>`. Table 3-5 shows the specialized functions and their uses.

We also need to call different functions on our RDD (so we can't just create a `DoubleFunction` and pass it to `map()`). When we want a `DoubleRDD` back, instead of calling `map()`, we need to call `mapToDouble()` with the same pattern all of the other functions follow.

Table 3-5. Java interfaces for type-specific functions

Function name	Equivalent function*<A, B,…>	Usage
`DoubleFlatMapFunction<T>`	`Function<T, Iterable<Double>>`	`DoubleRDD` from a `flatMapToDouble`
`DoubleFunction<T>`	`Function<T, double>`	`DoubleRDD` from `mapToDouble`
`PairFlatMapFunction<T, K, V>`	`Function<T, Iterable<Tuple2<K, V>>>`	`PairRDD<K, V>` from a `flatMapToPair`
`PairFunction<T, K, V>`	`Function<T, Tuple2<K, V>>`	`PairRDD<K, V>` from a `mapToPair`

We can modify Example 3-28, where we squared an RDD of numbers, to produce a `JavaDoubleRDD`, as shown in Example 3-38. This gives us access to the additional `DoubleRDD` specific functions like `mean()` and `variance()`.

Example 3-38. Creating DoubleRDD in Java

```
JavaDoubleRDD result = rdd.mapToDouble(
  new DoubleFunction<Integer>() {
    public double call(Integer x) {
      return (double) x * x;
    }
});
System.out.println(result.mean());
```

PYTHON

The Python API is structured differently than Java and Scala. In Python all of the functions are implemented on the base RDD class but will fail at runtime if the type of data in the RDD is incorrect.

Persistence (Caching)

pickled objects. When we write data out to disk or off-heap storage, that data is also always serialized.

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
MEMORY_ONLY	High	Low	Y	N
MEMORY_ONLY_SER	Low	High	Y	N
MEMORY_AND_DISK	High	Medium	Some	Some
MEMORY_AND_DISK_SER	Low	High	Some	Some
DISK_ONLY	Low	High	N	Y

> **TIP**
>
> Off-heap caching is experimental and uses Tachyon (http://Tachyon-project.org/). If you are interested in off-heap caching with Spark, take a look at the Running Spark on Tachyon guide (http://Tachyon-project.org/Running-Spark-on-Tachyon.html).

Example 3-40. persist() in Scala

```
val result = input.map(x => x * x)
result.persist(StorageLevel.DISK_ONLY)
println(result.count())
println(result.collect().mkString(","))
```

Notice that we called `persist()` on the RDD before the first action. The `persist()` call on its own doesn't force evaluation.

If you attempt to cache too much data to fit in memory, Spark will automatically evict old partitions using a Least Recently Used (LRU) cache policy. For the memory-only storage levels, it will recompute these partitions the next time they are accessed, while for the memory-and-disk ones, it will write them out to disk. In either case, this means that you don't have to worry about your job breaking if you ask Spark to cache too much data. However, caching unnecessary data can lead to eviction of useful data and more recomputation time.

Finally, RDDs come with a method called `unpersist()` that lets you manually remove them from the cache.

Conclusion

In this chapter, we have covered the RDD execution model and a large number of common operations on RDDs. If you have gotten here, congratulations—you've learned all the core concepts of working in Spark. In the next chapter, we'll cover a special set of operations available on RDDs of key/value pairs, which are the most common way to aggregate or group together data in parallel. After that, we discuss