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" on page 11). 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.[1]

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

```
>>> pythonLines.persist

>>> pythonLines.count()
2

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

To summarize, every Spark program and shell session will work as follows:

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

`cache()` is the same as calling `persist()` with the default 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.

1 The ability to always recompute an RDD is actually why RDDs are called "resilient." When a machine holding RDD data fails, Spark uses this ability to recompute the missing partitions, transparent to the user.

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

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

Example 3-6. parallelize() method in Scala

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

Example 3-7. parallelize() method in Java

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

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

Example 3-9. textFile() method in Scala

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

Example 3-10. textFile() method in Java

```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" on page 29, 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"))
```

Example 3-13. filter() transformation in Java

```
JavaRDD<String> inputRDD = sc.textFile("log.txt");
JavaRDD<String> errorsRDD = inputRDD.filter(
  new Function<String, Boolean>() {
    public Boolean call(String x) { return x.contains("error"); }
  }
});
```

Note that the `filter()` operation does not mutate the existing `inputRDD`. Instead, it returns a pointer to an entirely new RDD. `inputRDD` can still be reused later in the program—for instance, to search for other words. In fact, let's use `inputRDD` again to search for lines with the word *warning* in them. Then, we'll use another transformation, `union()`, to print out the number of lines that contained either *error* or *warning*. We show Python in Example 3-14, but the `union()` function is identical in all three languages.

Example 3-14. union() transformation in Python

```
errorsRDD = inputRDD.filter(lambda x: "error" in x)
warningsRDD = inputRDD.filter(lambda x: "warning" in x)
badLinesRDD = errorsRDD.union(warningsRDD)
```

`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.

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.

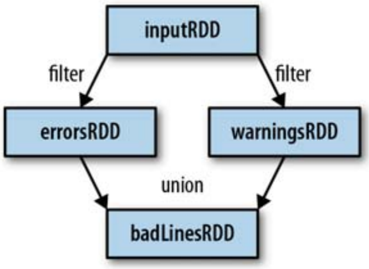


Figure 3-1. RDD lineage graph created during log analysis

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() + " concerning lines")
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.

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()`, or any of a number of actions for various built-in formats. We will cover the different options for exporting data in Chapter 5.

It is important to note that each time we call a new action, the entire RDD must be computed "from scratch." To avoid this inefficiency, users can *persist* intermediate results, as we will cover in "Persistence (Caching)" on page 44.

Lazy Evaluation

As you read earlier, transformations on RDDs are lazily evaluated, meaning that Spark will not begin to execute until it sees an action. This can be somewhat counter-intuitive for new users, but may be familiar for those who have used functional languages such as Haskell or LINQ-like data processing frameworks.

Lazy evaluation means that when we call a transformation on an RDD (for instance, calling `map()`), the operation is not immediately performed. Instead, Spark internally records metadata to indicate that this operation has been requested. Rather than thinking of an RDD as containing specific data, it is best to think of each RDD as consisting of instructions on how to compute the data that we build up through transformations. 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.

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

```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 variable
      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 it needs to be serializable (implementing Java's Serializable interface). Furthermore, as in Python, passing a method or field of an object includes a reference to that whole object, though this is less obvious because we are not forced to write these references with `self`. As we did with Python in Example 3-20, we can instead extract the fields we need as local variables and avoid needing to pass the whole object containing them, as shown in Example 3-21.

Example 3-21. Scala function passing

```
class SearchFunctions(val query: String) {
  def isMatch(s: String): Boolean = {
    s.contains(query)
  }
  def getMatchesFunctionReference(rdd: RDD[String]): RDD[Boolean] = {
    // Problem: "isMatch" means "this.isMatch", so we pass all of "this"
    rdd.map(isMatch)
  }
  def getMatchesFieldReference(rdd: RDD[String]): RDD[Array[String]] = {
    // Problem: "query" means "this.query", so we pass all of "this"
    rdd.map(x => x.split(query))
  }
  def getMatchesNoReference(rdd: RDD[String]): RDD[Array[String]] = {
    // Safe: extract just the field we need into a local variable
    val query_ = this.query
    rdd.map(x => x.split(query_))
  }
}
```

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" on page 43.

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

```
RDD<String> errors = lines.filter(s -> s.contains("error"));
```

If you are interested in using Java 8's lambda expression, refer to Oracle's documentation and the Databricks blog post on how to use lambdas with Spark.

Both anonymous inner classes and lambda expressions can reference any `final` variables in the method enclosing them, so you can pass these variables to Spark just as in Python and Scala.

Common Transformations and Actions

In this chapter, we tour the most common transformations and actions in Spark. Additional operations are available on RDDs containing certain types of data—for example, statistical functions on RDDs of numbers, and key/value operations such as 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.

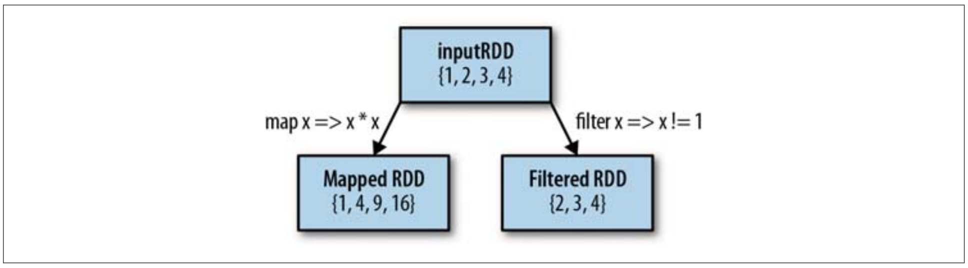


Figure 3-2. Mapped and filtered RDD from an input RDD

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"
```

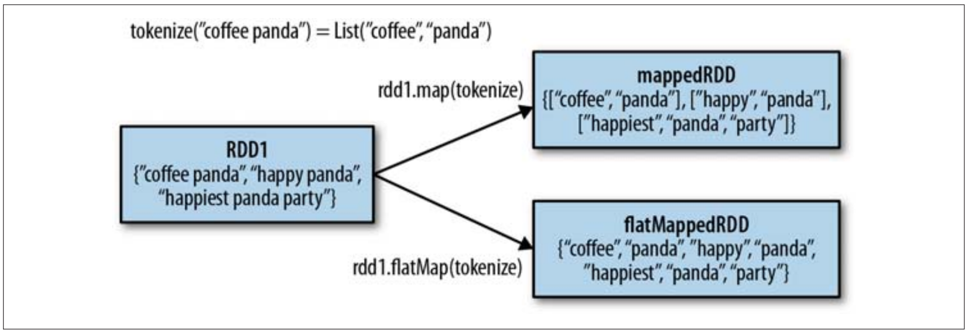
Example 3-30. flatMap() in Scala, splitting lines into multiple words

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

Example 3-31. flatMap() in Java, splitting lines into multiple words

```
JavaRDD<String> lines = sc.parallelize(Arrays.asList("hello world", "hi"));
JavaRDD<String> words = lines.flatMap(new FlatMapFunction<String, String>() {
  public Iterable<String> call(String line) {
    return Arrays.asList(line.split(" "));
  }
});
words.first();  // returns "hello"
```

We illustrate the difference between `flatMap()` and `map()` in Figure 3-3. You can think of `flatMap()` as "flattening" the iterators returned to it, so that instead of ending up with an RDD of lists we have an RDD of the elements in those lists.



tokenize("coffee panda") = List("coffee", "panda")
rdd1.map(tokenize)
mappedRDD
{["coffee", "panda"], ["happy", "panda"], ["happiest", "panda", "party"]}
RDD1
{"coffee panda", "happy panda", "happiest panda party"}
flatMappedRDD
{"coffee", "panda", "happy", "panda", "happiest", "panda", "party"}
rdd1.flatMap(tokenize)

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.

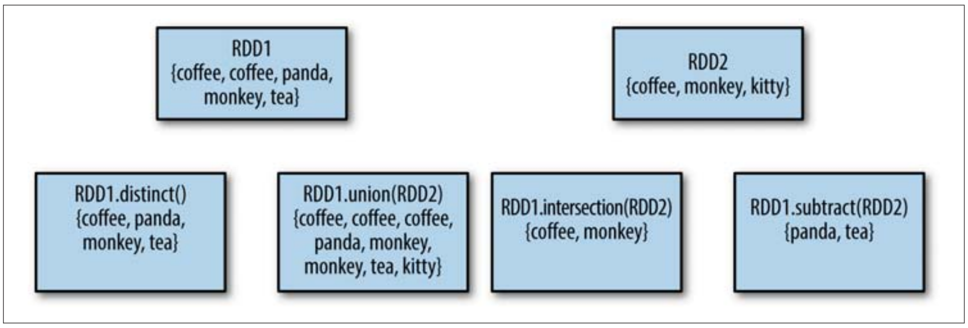


Figure 3-4. Some simple set operations

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.

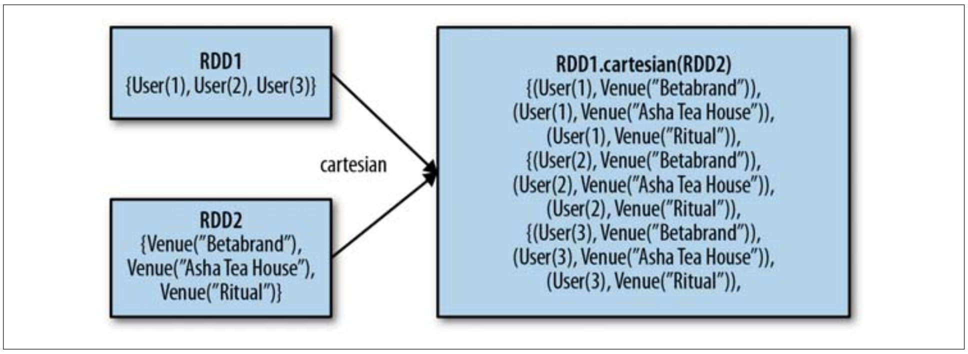

RDD1
{User(1), User(2), User(3)}
cartesian
RDD2
{Venue("Betabrand"), Venue("Asha Tea House"), Venue("Ritual")}
RDD1.cartesian(RDD2)
{(User(1), Venue("Betabrand")),
(User(1), Venue("Asha Tea House")),
(User(1), Venue("Ritual")),
{(User(2), Venue("Betabrand")),
(User(2), Venue("Asha Tea House")),
(User(2), Venue("Ritual")),
{(User(3), Venue("Betabrand")),
(User(3), Venue("Asha Tea House")),
(User(3), Venue("Ritual")),

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	Result
`map()`	Apply a function to each element in the RDD and return an RDD of the result.	`rdd.map(x => x + 1)`	`{2, 3, 4, 4}`
`flatMap()`	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.	`rdd.flatMap(x => x.to(3))`	`{1, 2, 3, 2, 3, 3, 3}`
`filter()`	Return an RDD consisting of only elements that pass the condition passed to `filter()`.	`rdd.filter(x => x != 1)`	`{2, 3, 3}`
`distinct()`	Remove duplicates.	`rdd.distinct()`	`{1, 2, 3}`
`sample(withReplacement, fraction, [seed])`	Sample an RDD, with or without replacement.	`rdd.sample(false, 0.5)`	Nondeterministic

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)
```

Example 3-33. reduce() in Scala

```
val sum = rdd.reduce((x, y) => x + y)
```

Example 3-34. reduce() in Java

```
Integer sum = rdd.reduce(new Function2<Integer, Integer, Integer>() {
  public Integer call(Integer x, Integer y) { return x + y; }
});
```

Similar to `reduce()` is `fold()`, which also takes a function with the same signature as needed for `reduce()`, but in addition takes a "zero value" to be used for the initial call on each partition. The zero value you provide should be the identity element for your operation; that is, applying it multiple times with your function should not change the value (e.g., 0 for +, 1 for *, or an empty list for concatenation).

> You can minimize object creation in `fold()` by modifying and returning the first of the two parameters in place. However, 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] + acc2[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 + acc2._2))
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 bottlenecks when you're dealing with large amounts of data.

If there is an ordering defined on our data, we can also extract the top elements from an RDD using `top()`. `top()` will use the default ordering on the data, but we can supply our own comparison function to extract the top elements.

Sometimes we need a sample of our data in our driver program. The `takeSample(withReplacement, num, seed)` function allows us to take a sample of our data either with or without replacement.

Sometimes it is useful to perform an action on all of the elements in the RDD, but without returning any result to the driver program. A good example of this would be posting JSON to a webserver or inserting records into a database. In either case, the `foreach()` action lets us perform computations on each element in the RDD without bringing it back locally.

The further standard operations on a basic RDD all behave pretty much exactly as you would imagine from their name. `count()` returns a count of the elements, and `countByValue()` returns a map of each unique value to its count. Table 3-4 summarizes these and other actions.

Table 3-4. Basic actions on an RDD containing {1, 2, 3, 3}

Function name	Purpose	Example	Result
`collect()`	Return all elements from the RDD.	`rdd.collect()`	`{1, 2, 3, 3}`
`count()`	Number of elements in the RDD.	`rdd.count()`	`4`
`countByValue()`	Number of times each element occurs in the RDD.	`rdd.countByValue()`	`{(1, 1), (2, 1), (3, 2)}`

Function name	Purpose	Example	Result
`take(num)`	Return `num` elements from the RDD.	`rdd.take(2)`	`{1, 2}`
`top(num)`	Return the top `num` elements the RDD.	`rdd.top(2)`	`{3, 3}`
`takeOrdered(num)(ordering)`	Return `num` elements based on provided ordering.	`rdd.takeOrdered(2)(myOrdering)`	`{3, 3}`
`takeSample(withReplacement, num, [seed])`	Return `num` elements at random.	`rdd.takeSample(false, 1)`	Nondeterministic
`reduce(func)`	Combine the elements of the RDD together in parallel (e.g., `sum`).	`rdd.reduce((x, y) => x + y)`	9
`fold(zero)(func)`	Same as `reduce()` but with the provided zero value.	`rdd.fold(0)((x, y) => x + y)`	9
`aggregate(zeroValue)(seqOp, combOp)`	Similar to `reduce()` but used to return a different type.	`rdd.aggregate((0, 0)) ((x, y) => (x._1 + y, x._2 + 1), (x, y) => (x._1 + y._1, x._2 + y._2))`	`(9, 4)`
`foreach(func)`	Apply the provided function to each element of the RDD.	`rdd.foreach(func)`	Nothing

Converting Between RDD Types

Some functions are available only on certain types of RDDs, such as `mean()` and `variance()` on numeric RDDs or `join()` on key/value pair RDDs. We will cover these special functions for numeric data in Chapter 6 and pair RDDs in Chapter 4. In Scala and Java, these methods aren't defined on the standard RDD class, so to access this additional functionality we have to make sure we get the correct specialized class.

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" on page 17, 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. 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 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`

Function name	Equivalent function*<A, B,...>	Usage
`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)

As discussed earlier, Spark RDDs are lazily evaluated, and sometimes we may wish to use the same RDD multiple times. If we do this naively, Spark will recompute the RDD and all of its dependencies each time we call an action on the RDD. This can be especially expensive for iterative algorithms, which look at the data many times. Another trivial example would be doing a count and then writing out the same RDD, as shown in Example 3-39.

Example 3-39. Double execution in Scala

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

To avoid computing an RDD multiple times, we can ask Spark to *persist* the data. When we ask Spark to persist an RDD, the nodes that compute the RDD store their partitions. If a node that has data persisted on it fails, Spark will recompute the lost partitions of the data when needed. We can also replicate our data on multiple nodes if we want to be able to handle node failure without slowdown.

Spark has many levels of persistence to choose from based on what our goals are, as you can see in Table 3-6. In Scala (Example 3-40) and Java, the default `persist()` will store the data in the JVM heap as unserialized objects. In Python, we always serialize the data that persist stores, so the default is instead stored in the JVM heap as 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	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	

Off-heap caching is experimental and uses Tachyon. If you are interested in off-heap caching with Spark, take a look at the Running Spark on Tachyon guide.

Example 3-40. persist() in Scala

```
import org.apache.spark.storage.StorageLevel
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 input and output from a variety of data sources, and more advanced topics in working with SparkContext.

CHAPTER 4

Working with Key/Value Pairs

This chapter covers how to work with RDDs of key/value pairs, which are a common data type required for many operations in Spark. Key/value RDDs are commonly used to perform aggregations, and often we will do some initial ETL (extract, transform, and load) to get our data into a key/value format. Key/value RDDs expose new operations (e.g., counting up reviews for each product, grouping together data with the same key, and grouping together two different RDDs).

We also discuss an advanced feature that lets users control the layout of pair RDDs across nodes: *partitioning*. Using controllable partitioning, applications can sometimes greatly reduce communication costs by ensuring that data will be accessed together and will be on the same node. This can provide significant speedups. We illustrate partitioning using the PageRank algorithm as an example. Choosing the right partitioning for a distributed dataset is similar to choosing the right data structure for a local one—in both cases, data layout can greatly affect performance.

Motivation

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.

Creating Pair RDDs

There are a number of ways to get pair RDDs in Spark. Many formats we explore loading from in Chapter 5 will directly return pair RDDs for their key/value data. In other cases we have a regular RDD that we want to turn into a pair RDD. We can do this by running a `map()` function that returns key/value pairs. To illustrate, we show code that starts with an RDD of lines of text and keys the data by the first word in each line.

The way to build key-value RDDs differs by language. In Python, for the functions on keyed data to work we need to return an RDD composed of tuples (see Example 4-1).

Example 4-1. Creating a pair RDD using the first word as the key in Python

```
pairs = lines.map(lambda x: (x.split(" ")[0], x))
```

In Scala, for the functions on keyed data to be available, we also need to return tuples (see Example 4-2). An implicit conversion on RDDs of tuples exists to provide the additional key/value functions.

Example 4-2. Creating a pair RDD using the first word as the key in Scala

```
val pairs = lines.map(x => (x.split(" ")(0), x))
```

Java doesn't have a built-in tuple type, so Spark's Java API has users create tuples using the `scala.Tuple2` class. This class is very simple: Java users can construct a new tuple by writing `new Tuple2(elem1, elem2)` and can then access its elements with the `._1()` and `._2()` methods.

Java users also need to call special versions of Spark's functions when creating pair RDDs. For instance, the `mapToPair()` function should be used in place of the basic `map()` function. This is discussed in more detail in "Java" on page 43, but let's look at a simple case in Example 4-3.

Example 4-3. Creating a pair RDD using the first word as the key in Java

```
PairFunction<String, String, String> keyData =
  new PairFunction<String, String, String>() {
  public Tuple2<String, String> call(String x) {
    return new Tuple2(x.split(" ")[0], x);
  }
};
JavaPairRDD<String, String> pairs = lines.mapToPair(keyData);
```

When creating a pair RDD from an in-memory collection in Scala and Python, we only need to call `SparkContext.parallelize()` on a collection of pairs. To create a pair RDD in Java from an in-memory collection, we instead use `SparkContext.parallelizePairs()`.

Transformations on Pair RDDs

Pair RDDs are allowed to use all the transformations available to standard RDDs. The same rules apply from "Passing Functions to Spark" on page 30. Since pair RDDs contain tuples, we need to pass functions that operate on tuples rather than on individual elements. Tables 4-1 and 4-2 summarize transformations on pair RDDs, and we will dive into the transformations in detail later in the chapter.

Table 4-1. Transformations on one pair RDD (example: {(1, 2), (3, 4), (3, 6)})

Function name	Purpose	Example	Result
`reduceByKey(func)`	Combine values with the same key.	`rdd.reduceByKey( (x, y) => x + y)`	`{(1, 2), (3, 10)}`
`groupByKey()`	Group values with the same key.	`rdd.groupByKey()`	`{(1, [2]), (3, [4, 6])}`
`combineByKey(createCombiner, mergeValue, mergeCombiners, partitioner)`	Combine values with the same key using a different result type.	See Examples 4-12 through 4-14.	
`mapValues(func)`	Apply a function to each value of a pair RDD without changing the key.	`rdd.mapValues(x => x+1)`	`{(1, 3), (3, 5), (3, 7)}`
`flatMapValues(func)`	Apply a function that returns an iterator to each value of a pair RDD, and for each element returned, produce a key/value entry with the old key. Often used for tokenization.	`rdd.flatMapValues(x => (x to 5)`	`{(1, 2), (1, 3), (1, 4), (1, 5), (3, 4), (3, 5)}`
`keys()`	Return an RDD of just the keys.	`rdd.keys()`	`{1, 3, 3}`

Function name	Purpose	Example	Result
`values()`	Return an RDD of just the values.	`rdd.values()`	`{2, 4, 6}`
`sortByKey()`	Return an RDD sorted by the key.	`rdd.sortByKey()`	`{(1, 2), (3, 4), (3, 6)}`

Table 4-2. Transformations on two pair RDDs (rdd = {(1, 2), (3, 4), (3, 6)} other = {(3, 9)})

Function name	Purpose	Example	Result
`subtractByKey`	Remove elements with a key present in the other RDD.	`rdd.subtractByKey(other)`	`{(1, 2)}`
`join`	Perform an inner join between two RDDs.	`rdd.join(other)`	`{(3, (4, 9)), (3, (6, 9))}`
`rightOuterJoin`	Perform a join between two RDDs where the key must be present in the other RDD.	`rdd.rightOuterJoin(other)`	`{(3,(Some(4),9)), (3,(Some(6),9))}`
`leftOuterJoin`	Perform a join between two RDDs where the key must be present in the first RDD.	`rdd.leftOuterJoin(other)`	`{(1,(2,None)), (3, (4,Some(9))), (3, (6,Some(9)))}`
`cogroup`	Group data from both RDDs sharing the same key.	`rdd.cogroup(other)`	`{(1,([2],[])), (3, ([4, 6],[9]))}`

We discuss each of these families of pair RDD functions in more detail in the upcoming sections.

Pair RDDs are also still RDDs (of Tuple2 objects in Java/Scala or of Python tuples), and thus support the same functions as RDDs. For instance, we can take our pair RDD from the previous section and filter out lines longer than 20 characters or more, as shown in Examples 4-4 through 4-6 and Figure 4-1.

Example 4-4. Simple filter on second element in Python

```python
result = pairs.filter(lambda keyValue: len(keyValue[1]) < 20)
```

Example 4-5. Simple filter on second element in Scala

```scala
pairs.filter{case (key, value) => value.length < 20}
```

Example 4-6. Simple filter on second element in Java

```
Function<Tuple2<String, String>, Boolean> longWordFilter =
  new Function<Tuple2<String, String>, Boolean>() {
    public Boolean call(Tuple2<String, String> keyValue) {
      return (keyValue._2().length() < 20);
    }
  };
JavaPairRDD<String, String> result = pairs.filter(longWordFilter);
```

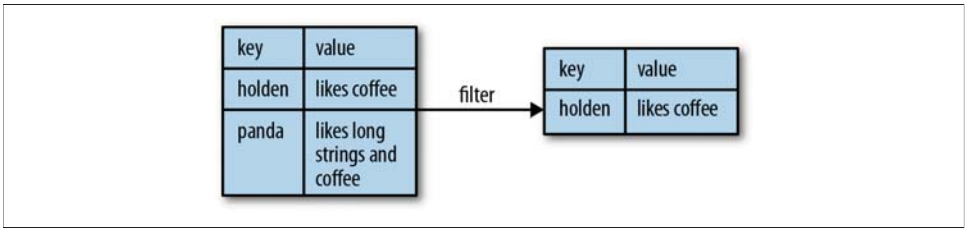


Figure 4-1. Filter on value

Sometimes working with pairs can be awkward if we want to access only the value part of our pair RDD. Since this is a common pattern, Spark provides the `mapValues(func)` function, which is the same as `map{case (x, y): (x, func(y))}`. We will use this function in many of our examples.

We now discuss each of the families of pair RDD functions, starting with aggregations.

Aggregations

When datasets are described in terms of key/value pairs, it is common to want to aggregate statistics across all elements with the same key. We have looked at the `fold()`, `aggregate()`, and `reduce()` actions on basic RDDs, and similar per-key transformations exist on pair RDDs. Spark has a similar set of operations that combines values that have the same key. These operations return RDDs and thus are transformations rather than actions.

`reduceByKey()` is quite similar to `reduce()`; both take a function and use it to combine values. `reduceByKey()` runs several parallel reduce operations, one for each key in the dataset, where each operation combines values that have the same key. Because datasets can have very large numbers of keys, `reduceByKey()` is not implemented as an action that returns a value to the user program. Instead, it returns a new RDD consisting of each key and the reduced value for that key.

`foldByKey()` is quite similar to `fold()`; both use a zero value of the same type of the data in our RDD and combination function. As with `fold()`, the provided zero value for `foldByKey()` should have no impact when added with your combination function to another element.

As Examples 4-7 and 4-8 demonstrate, we can use `reduceByKey()` along with `mapValues()` to compute the per-key average in a very similar manner to how `fold()` and `map()` can be used to compute the entire RDD average (see Figure 4-2). As with averaging, we can achieve the same result using a more specialized function, which we will cover next.

Example 4-7. Per-key average with reduceByKey() and mapValues() in Python

```
rdd.mapValues(lambda x: (x, 1)).reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1]))
```

Example 4-8. Per-key average with reduceByKey() and mapValues() in Scala

```
rdd.mapValues(x => (x, 1)).reduceByKey((x, y) => (x._1 + y._1, x._2 + y._2))
```

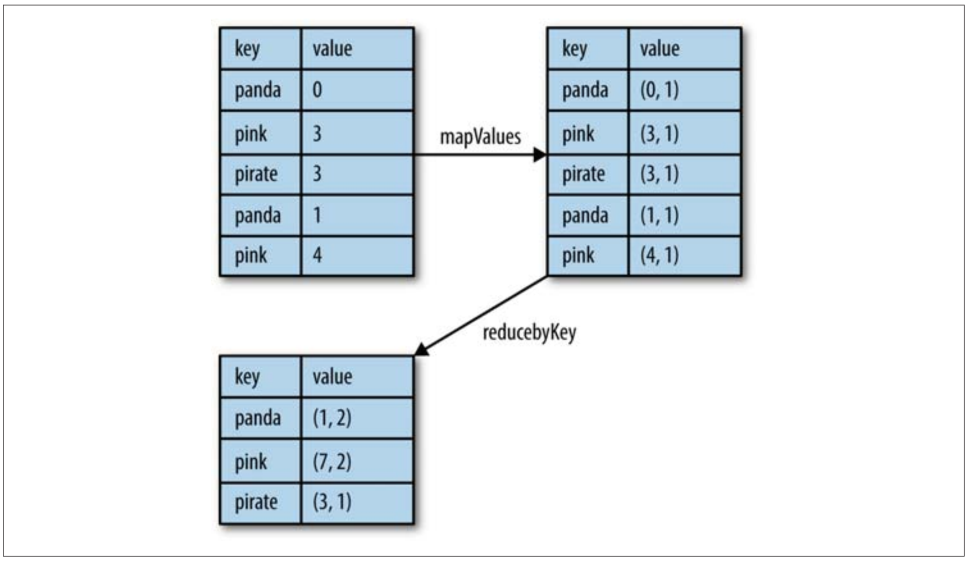


Figure 4-2. Per-key average data flow

Those familiar with the combiner concept from MapReduce should note that calling `reduceByKey()` and `foldByKey()` will automatically perform combining locally on each machine before computing global totals for each key. The user does not need to specify a combiner. The more general `combineByKey()` interface allows you to customize combining behavior.

We can use a similar approach in Examples 4-9 through 4-11 to also implement the classic distributed word count problem. We will use `flatMap()` from the previous chapter so that we can produce a pair RDD of words and the number 1 and then sum together all of the words using `reduceByKey()` as in Examples 4-7 and 4-8.

Example 4-9. Word count in Python

```python
rdd = sc.textFile("s3://...")
words = rdd.flatMap(lambda x: x.split(" "))
result = words.map(lambda x: (x, 1)).reduceByKey(lambda x, y: x + y)
```

Example 4-10. Word count in Scala

```scala
val input = sc.textFile("s3://...")
val words = input.flatMap(x => x.split(" "))
val result = words.map(x => (x, 1)).reduceByKey((x, y) => x + y)
```

Example 4-11. Word count in Java

```java
JavaRDD<String> input = sc.textFile("s3://...")
JavaRDD<String> words = input.flatMap(new FlatMapFunction<String, String>() {
  public Iterable<String> call(String x) { return Arrays.asList(x.split(" ")); }
});
JavaPairRDD<String, Integer> result = words.mapToPair(
  new PairFunction<String, String, Integer>() {
    public Tuple2<String, Integer> call(String x) { return new Tuple2(x, 1); }
}).reduceByKey(
  new Function2<Integer, Integer, Integer>() {
    public Integer call(Integer a, Integer b) { return a + b; }
});
```

We can actually implement word count even faster by using the `countByValue()` function on the first RDD: `input.flatMap(x => x.split(" ")).countByValue()`.

`combineByKey()` is the most general of the per-key aggregation functions. Most of the other per-key combiners are implemented using it. Like `aggregate()`, `combineByKey()` allows the user to return values that are not the same type as our input data.

To understand `combineByKey()`, it's useful to think of how it handles each element it processes. As `combineByKey()` goes through the elements in a partition, each element either has a key it hasn't seen before or has the same key as a previous element.

If it's a new element, `combineByKey()` uses a function we provide, called `createCombiner()`, to create the initial value for the accumulator on that key. It's important to note that this happens the first time a key is found in each partition, rather than only the first time the key is found in the RDD.

If it is a value we have seen before while processing that partition, it will instead use the provided function, `mergeValue()`, with the current value for the accumulator for that key and the new value.

Since each partition is processed independently, we can have multiple accumulators for the same key. When we are merging the results from each partition, if two or more partitions have an accumulator for the same key we merge the accumulators using the user-supplied `mergeCombiners()` function.

> We can disable map-side aggregation in `combineByKey()` if we know that our data won't benefit from it. For example, `groupByKey()` disables map-side aggregation as the aggregation function (appending to a list) does not save any space. If we want to disable map-side combines, we need to specify the partitioner; for now you can just use the partitioner on the source RDD by passing `rdd.partitioner`.

Since `combineByKey()` has a lot of different parameters it is a great candidate for an explanatory example. To better illustrate how `combineByKey()` works, we will look at computing the average value for each key, as shown in Examples 4-12 through 4-14 and illustrated in Figure 4-3.

Example 4-12. Per-key average using combineByKey() in Python

```python
sumCount = nums.combineByKey((lambda x: (x,1)),
                             (lambda x, y: (x[0] + y, x[1] + 1)),
                             (lambda x, y: (x[0] + y[0], x[1] + y[1])))
r = sumCount.map(lambda key, xy: (key, xy[0]/xy[1])).collectAsMap()
print(*r)
```

Example 4-13. Per-key average using combineByKey() in Scala

```scala
val result = input.combineByKey(
  (v) => (v, 1),
  (acc: (Int, Int), v) => (acc._1 + v, acc._2 + 1),
  (acc1: (Int, Int), acc2: (Int, Int)) => (acc1._1 + acc2._1, acc1._2 + acc2._2)
  ).map{ case (key, value) => (key, value._1 / value._2.toFloat) }
  result.collectAsMap().foreach(println(_))
```

Example 4-14. Per-key average using combineByKey() in Java

```java
public static class AvgCount implements Serializable {
  public AvgCount(int total, int num) {   total_ = total;   num_ = num; }
  public int total_;
  public int num_;
  public float avg() {   return total_ / (float) num_; }
}

Function<Integer, AvgCount> createAcc = new Function<Integer, AvgCount>() {
  public AvgCount call(Integer x) {
    return new AvgCount(x, 1);
  }
};
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);
JavaPairRDD<String, AvgCount> avgCounts =
  nums.combineByKey(createAcc, addAndCount, combine);
Map<String, AvgCount> countMap = avgCounts.collectAsMap();
for (Entry<String, AvgCount> entry : countMap.entrySet()) {
  System.out.println(entry.getKey() + ":" + entry.getValue().avg());
}
```

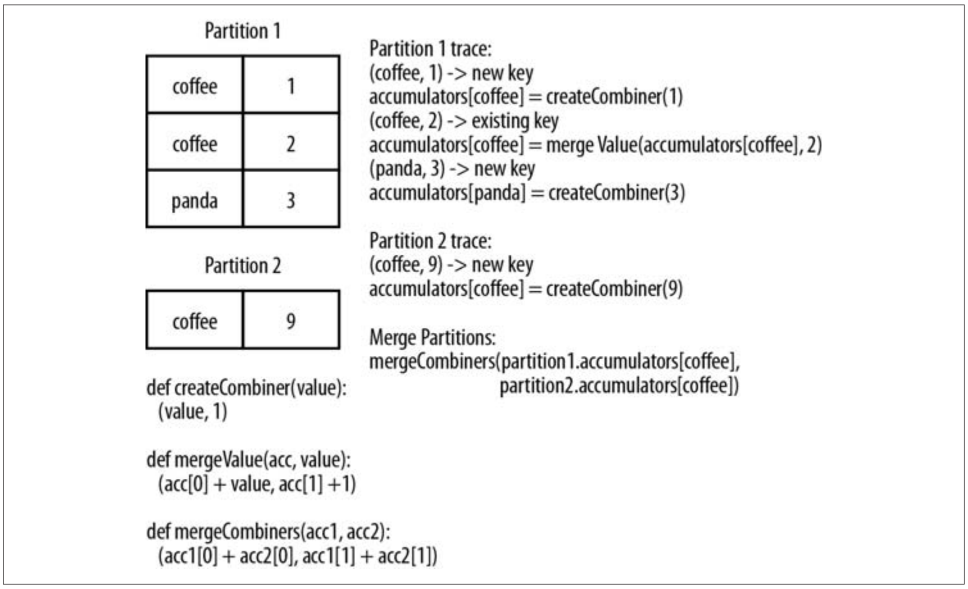

Partition 1
coffee 1
coffee 2
panda 3
Partition 2
coffee 9
def createCombiner(value):
  (value, 1)
def mergeValue(acc, value):
  (acc[0] + value, acc[1] +1)
def mergeCombiners(acc1, acc2):
  (acc1[0] + acc2[0], acc1[1] + acc2[1])
Partition 1 trace:
(coffee, 1) -> new key
accumulators[coffee] = createCombiner(1)
(coffee, 2) -> existing key
accumulators[coffee] = merge Value(accumulators[coffee], 2)
(panda, 3) -> new key
accumulators[panda] = createCombiner(3)
Partition 2 trace:
(coffee, 9) -> new key
accumulators[coffee] = createCombiner(9)
Merge Partitions:
mergeCombiners(partition1.accumulators[coffee],
partition2.accumulators[coffee])

Figure 4-3. combineByKey() sample data flow

There are many options for combining our data by key. Most of them are implemented on top of `combineByKey()` but provide a simpler interface. In any case, using one of the specialized aggregation functions in Spark can be much faster than the naive approach of grouping our data and then reducing it.

Tuning the level of parallelism

So far we have talked about how all of our transformations are distributed, but we have not really looked at how Spark decides how to split up the work. Every RDD has a fixed number of *partitions* that determine the degree of parallelism to use when executing operations on the RDD.

When performing aggregations or grouping operations, we can ask Spark to use a specific number of partitions. Spark will always try to infer a sensible default value based on the size of your cluster, but in some cases you will want to tune the level of parallelism for better performance.

Most of the operators discussed in this chapter accept a second parameter giving the number of partitions to use when creating the grouped or aggregated RDD, as shown in Examples 4-15 and 4-16.

Example 4-15. reduceByKey() with custom parallelism in Python

```
data = [("a", 3), ("b", 4), ("a", 1)]
sc.parallelize(data).reduceByKey(lambda x, y: x + y)      # Default parallelism
sc.parallelize(data).reduceByKey(lambda x, y: x + y, 10)  # Custom parallelism
```

Example 4-16. reduceByKey() with custom parallelism in Scala

```
val data = Seq(("a", 3), ("b", 4), ("a", 1))
sc.parallelize(data).reduceByKey((x, y) => x + y)     // Default parallelism
sc.parallelize(data).reduceByKey((x, y) => x + y, 10) // Custom parallelism
```

Sometimes, we want to change the partitioning of an RDD outside the context of grouping and aggregation operations. For those cases, Spark provides the `repartition()` function, which shuffles the data across the network to create a new set of partitions. Keep in mind that repartitioning your data is a fairly expensive operation. Spark also has an optimized version of `repartition()` called `coalesce()` that allows avoiding data movement, but only if you are decreasing the number of RDD partitions. To know whether you can safely call `coalesce()`, you can check the size of the RDD using `rdd.partitions.size()` in Java/Scala and `rdd.getNumPartitions()` in Python and make sure that you are coalescing it to fewer partitions than it currently has.

Grouping Data

With keyed data a common use case is grouping our data by key—for example, viewing all of a customer's orders together.

If our data is already keyed in the way we want, `groupByKey()` will group our data using the key in our RDD. On an RDD consisting of keys of type `K` and values of type `V`, we get back an RDD of type `[K, Iterable[V]]`.

`groupBy()` works on unpaired data or data where we want to use a different condition besides equality on the current key. It takes a function that it applies to every element in the source RDD and uses the result to determine the key.

If you find yourself writing code where you `groupByKey()` and then use a `reduce()` or `fold()` on the values, you can probably achieve the same result more efficiently by using one of the per-key aggregation functions. Rather than reducing the RDD to an in-memory value, we reduce the data per key and get back an RDD with the reduced values corresponding to each key. For example, `rdd.reduceByKey(func)` produces the same RDD as `rdd.groupByKey().mapValues(value => value.reduce(func))` but is more efficient as it avoids the step of creating a list of values for each key.

In addition to grouping data from a single RDD, we can group data sharing the same key from multiple RDDs using a function called `cogroup()`. `cogroup()` over two RDDs sharing the same key type, `K`, with the respective value types `V` and `W` gives us back `RDD[(K, (Iterable[V], Iterable[W]))]`. If one of the RDDs doesn't have elements for a given key that is present in the other RDD, the corresponding `Iterable` is simply empty. `cogroup()` gives us the power to group data from multiple RDDs.

`cogroup()` is used as a building block for the joins we discuss in the next section.

`cogroup()` can be used for much more than just implementing joins. We can also use it to implement intersect by key. Additionally, `cogroup()` can work on three or more RDDs at once.

Joins

Some of the most useful operations we get with keyed data comes from using it together with other keyed data. Joining data together is probably one of the most common operations on a pair RDD, and we have a full range of options including right and left outer joins, cross joins, and inner joins.

The simple `join` operator is an inner join.[1] Only keys that are present in both pair RDDs are output. When there are multiple values for the same key in one of the inputs, the resulting pair RDD will have an entry for every possible pair of values with that key from the two input RDDs. A simple way to understand this is by looking at Example 4-17.

Example 4-17. Scala shell inner join

```
storeAddress = {
  (Store("Ritual"), "1026 Valencia St"), (Store("Philz"), "748 Van Ness Ave"),
  (Store("Philz"), "3101 24th St"), (Store("Starbucks"), "Seattle")}

storeRating = {
  (Store("Ritual"), 4.9), (Store("Philz"), 4.8))}

storeAddress.join(storeRating) == {
  (Store("Ritual"), ("1026 Valencia St", 4.9)),
  (Store("Philz"), ("748 Van Ness Ave", 4.8)),
  (Store("Philz"), ("3101 24th St", 4.8))}
```

1 "Join" is a database term for combining fields from two tables using common values.

Sometimes we don't need the key to be present in both RDDs to want it in our result. For example, if we were joining customer information with recommendations we might not want to drop customers if there were not any recommendations yet. `leftOuterJoin(other)` and `rightOuterJoin(other)` both join pair RDDs together by key, where one of the pair RDDs can be missing the key.

With `leftOuterJoin()` the resulting pair RDD has entries for each key in the source RDD. The value associated with each key in the result is a tuple of the value from the source RDD and an `Option` (or `Optional` in Java) for the value from the other pair RDD. In Python, if a value isn't present `None` is used; and if the value is present the regular value, without any wrapper, is used. As with `join()`, we can have multiple entries for each key; when this occurs, we get the Cartesian product between the two lists of values.

`Optional` is part of Google's Guava library and represents a possibly missing value. We can check `isPresent()` to see if it's set, and `get()` will return the contained instance provided data is present.

`rightOuterJoin()` is almost identical to `leftOuterJoin()` except the key must be present in the other RDD and the tuple has an option for the source rather than the other RDD.

We can revisit Example 4-17 and do a `leftOuterJoin()` and a `rightOuterJoin()` between the two pair RDDs we used to illustrate `join()` in Example 4-18.

Example 4-18. leftOuterJoin() and rightOuterJoin()

```
storeAddress.leftOuterJoin(storeRating) ==
{(Store("Ritual"),("1026 Valencia St",Some(4.9))),
  (Store("Starbucks"),("Seattle",None)),
  (Store("Philz"),("748 Van Ness Ave",Some(4.8))),
  (Store("Philz"),("3101 24th St",Some(4.8)))}

storeAddress.rightOuterJoin(storeRating) ==
{(Store("Ritual"),(Some("1026 Valencia St"),4.9)),
  (Store("Philz"),(Some("748 Van Ness Ave"),4.8)),
  (Store("Philz"), (Some("3101 24th St"),4.8))}
```

Sorting Data

Having sorted data is quite useful in many cases, especially when you're producing downstream output. We can sort an RDD with key/value pairs provided that there is an ordering defined on the key. Once we have sorted our data, any subsequent call on the sorted data to `collect()` or `save()` will result in ordered data.

Since we often want our RDDs in the reverse order, the `sortByKey()` function takes a parameter called `ascending` indicating whether we want it in ascending order (it defaults to `true`). Sometimes we want a different sort order entirely, and to support this we can provide our own comparison function. In Examples 4-19 through 4-21, we will sort our RDD by converting the integers to strings and using the string comparison functions.

Example 4-19. Custom sort order in Python, sorting integers as if strings

```
rdd.sortByKey(ascending=True, numPartitions=None, keyfunc = lambda x: str(x))
```

Example 4-20. Custom sort order in Scala, sorting integers as if strings

```
val input: RDD[(Int, Venue)] = ...
implicit val sortIntegersByString = new Ordering[Int] {
  override def compare(a: Int, b: Int) = a.toString.compare(b.toString)
}
rdd.sortByKey()
```

Example 4-21. Custom sort order in Java, sorting integers as if strings

```
class IntegerComparator implements Comparator<Integer> {
   public int compare(Integer a, Integer b) {
     return String.valueOf(a).compareTo(String.valueOf(b))
  }
}
rdd.sortByKey(comp)
```

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