**Chapter 12. Resilient Distributed Datasets (RDDs)**

The previous part of the book covered Spark’s Structured APIs. You should heavily favor these APIs in almost all scenarios. That being said, there are times when higher-level manipulation will not meet the business or engineering problem you are trying to solve. For those cases, you might need to use Spark’s lower-level APIs, specifically the Resilient Distributed Dataset (RDD), the SparkContext, and distributed *shared variables* like accumulators and broadcast variables. The chapters that follow in this part cover these APIs and how to use them.

**WARNING**

If you are brand new to Spark, this is not the place to start. Start with the Structured APIs, you’ll be more productive more quickly!

**What Are the Low-Level APIs?**

There are two sets of low-level APIs: there is one for manipulating distributed data (RDDs), and another for distributing and manipulating distributed shared variables (broadcast variables and accumulators).

**When to Use the Low-Level APIs?**

You should generally use the lower-level APIs in three situations:

* You need some functionality that you cannot find in the higher-level APIs; for example, if you need very tight control over physical data placement across the cluster.
* You need to maintain some legacy codebase written using RDDs.
* You need to do some custom shared variable manipulation. We will discuss shared variables more in [Chapter 14](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch14.html#s3c3---distributed-variables).

Those are the reasons why you should *use* these lower-level tools, buts it’s still helpful to *understand* these tools because all Spark workloads compile down to these fundamental primitives. When you’re calling a DataFrame transformation, it actually just becomes a set of RDD transformations. This understanding can make your task easier as you begin debugging more and more complex workloads.

Even if you are an advanced developer hoping to get the most out of Spark, we still recommend focusing on the Structured APIs. However, there are times when you might want to “drop down” to some of the lower-level tools to complete your task. You might need to drop down to these APIs to use some legacy code, implement some custom partitioner, or update and track the value of a variable over the course of a data pipeline’s execution. These tools give you more fine-grained control at the expense of safeguarding you from shooting yourself in the foot.

**How to Use the Low-Level APIs?**

A SparkContext is the entry point for low-level API functionality. You access it through the SparkSession, which is the tool you use to perform computation across a Spark cluster. We discuss this further in [Chapter 15](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch15.html#s4c0---how-spark-runs-on-a-cluster) but for now, you simply need to know that you can access a SparkContext via the following call:

spark.sparkContext

**About RDDs**

RDDs were the primary API in the Spark 1.X series and are still available in 2.X, but they are not as commonly used. However, as we’ve pointed out earlier in this book, virtually all Spark code you run, whether DataFrames or Datasets, compiles down to an RDD. The Spark UI, covered in the next part of the book, also describes job execution in terms of RDDs. Therefore, it will behoove you to have at least a basic understanding of what an RDD is and how to use it.

In short, an RDD represents an immutable, partitioned collection of records that can be operated on in parallel. Unlike DataFrames though, where each record is a structured row containing fields with a known schema, in RDDs the records are just Java, Scala, or Python objects of the programmer’s choosing.

RDDs give you complete control because every record in an RDD is a just a Java or Python object. You can store anything you want in these objects, in any format you want. This gives you great power, but not without potential issues. Every manipulation and interaction between values must be defined by hand, meaning that you must “reinvent the wheel” for whatever task you are trying to carry out. Also, optimizations are going to require much more manual work, because Spark does not understand the inner structure of your records as it does with the Structured APIs. For instance, Spark’s Structured APIs automatically store data in an optimzied, compressed binary format, so to achieve the same space-efficiency and performance, you’d also need to implement this type of format inside your objects and all the low-level operations to compute over it. Likewise, optimizations like reordering filters and aggregations that occur automatically in Spark SQL need to be implemented by hand. For this reason and others, we highly recommend using the Spark Structured APIs when possible.

The RDD API is similar to the Dataset, which we saw in the previous part of the book, except that RDDs are not stored in, or manipulated with, the structured data engine. However, it is trivial to convert back and forth between RDDs and Datasets, so you can use both APIs to take advantage of each API’s strengths and weaknesses. We’ll show how to do this throughout this part of the book.

**Types of RDDs**

If you look through Spark’s API documentation, you will notice that there are lots of subclasses of RDD. For the most part, these are internal representations that the DataFrame API uses to create optimized physical execution plans. As a user, however, you will likely only be creating two types of RDDs: the “generic” RDD type or a key-value RDD that provides additional functions, such as aggregating by key. For your purposes, these will be the only two types of RDDs that matter. Both just represent a collection of objects, but key-value RDDs have special operations as well as a concept of custom partitioning by key.

Let’s formally define RDDs. Internally, each RDD is characterized by five main properties:

* A list of partitions
* A function for computing each split
* A list of dependencies on other RDDs
* Optionally, a Partitioner for key-value RDDs (e.g., to say that the RDD is hash-partitioned)
* Optionally, a list of preferred locations on which to compute each split (e.g., block locations for a Hadoop Distributed File System [HDFS] file)

**NOTE**

The Partitioner is probably one of the core reasons why you might want to use RDDs in your code. Specifying your own custom Partitioner can give you significant performance and stability improvements if you use it correctly. This is discussed in more depth in [Chapter 13](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch13.html#s3c2---advanced-rdds) when we introduce Key–Value Pair RDDs.

These properties determine all of Spark’s ability to schedule and execute the user program. Different kinds of RDDs implement their own versions of each of the aforementioned properties, allowing you to define new data sources.

RDDs follow the exact same Spark programming paradigms that we saw in earlier chapters. They provide *transformations*, which evaluate lazily, and *actions*, which evaluate eagerly, to manipulate data in a distributed fashion. These work the same way as transformations and actions on DataFrames and Datasets. However, there is no concept of “rows” in RDDs; individual records are just raw Java/Scala/Python objects, and you manipulate those manually instead of tapping into the repository of functions that you have in the structured APIs.

The RDD APIs are available in Python as well as Scala and Java. For Scala and Java, the performance is for the most part the same, the large costs incurred in manipulating the raw objects. Python, however, can lose a substantial amount of performance when using RDDs. Running Python RDDs equates to running Python user-defined functions (UDFs) row by row. Just as we saw in [Chapter 6](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch06.html#s2c3---working-with-different-types-of-data). We serialize the data to the Python process, operate on it in Python, and then serialize it back to the Java Virtual Machine (JVM). This causes a high overhead for Python RDD manipulations. Even though many people ran production code with them in the past, we recommend building on the Structured APIs in Python and only dropping down to RDDs if absolutely necessary.

**When to Use RDDs?**

In general, you should not manually create RDDs unless you have a very, very specific reason for doing so. They are a much lower-level API that provides a lot of power but also lacks a lot of the optimizations that are available in the Structured APIs. For the vast majority of use cases, DataFrames will be more efficient, more stable, and more expressive than RDDs.

The most likely reason for why you’ll want to use RDDs is because you need fine-grained control over the physical distribution of data (custom partitioning of data).

**Datasets and RDDs of Case Classes**

We noticed this question on the web and found it to be an interesting one: what is the difference between RDDs of Case Classes and Datasets? The difference is that Datasets can still take advantage of the wealth of functions and optimizations that the Structured APIs have to offer. With Datasets, you do not need to choose between only operating on JVM types or on Spark types, you can choose whatever is either easiest to do or most flexible. You get the both of best worlds.

**Creating RDDs**

Now that we discussed some key RDD properties, let’s begin applying them so that you can better understand how to use them.

**Interoperating Between DataFrames, Datasets, and RDDs**

One of the easiest ways to get RDDs is from an existing DataFrame or Dataset. Converting these to an RDD is simple: just use the rdd method on any of these data types. You’ll notice that if you do a conversion from a Dataset[T] to an RDD, you’ll get the appropriate native type T back (remember this applies only to Scala and Java):

*// in Scala: converts a Dataset[Long] to RDD[Long]*

spark.range(500).rdd

Because Python doesn’t have Datasets—it has only DataFrames—you will get an RDD of type Row:

*# in Python*

spark.range(10).rdd

To operate on this data, you will need to convert this Row object to the correct data type or extract values out of it, as shown in the example that follows. This is now an RDD of type Row:

*// in Scala*

spark.range(10).toDF().rdd.map(rowObject **=>** rowObject.getLong(0))

*# in Python*

spark.range(10).toDF("id").rdd.map(**lambda** row: row[0])

You can use the same methodology to create a DataFrame or Dataset from an RDD. All you need to do is call the toDF method on the RDD:

*// in Scala*

spark.range(10).rdd.toDF()

*# in Python*

spark.range(10).rdd.toDF()

This command creates an RDD of type Row. This row is the internal Catalyst format that Spark uses to represent data in the Structured APIs. This functionality makes it possible for you to jump between the Structured and low-level APIs as it suits your use case. (We talk about this in [Chapter 13](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch13.html#s3c2---advanced-rdds).)

The RDD API will feel quite similar to the Dataset API in [Chapter 11](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch11.html#s2c8---datasets) because they are extremely similar to each other (RDDs being a lower-level representation of Datasets) that do not have a lot of the convenient functionality and interfaces that the Structured APIs do.

**From a Local Collection**

To create an RDD from a collection, you will need to use the parallelize method on a SparkContext (within a SparkSession). This turns a single node collection into a parallel collection. When creating this parallel collection, you can also explicitly state the number of partitions into which you would like to distribute this array. In this case, we are creating two partitions:

*// in Scala*

**val** myCollection **=** "Spark The Definitive Guide : Big Data Processing Made Simple"

.split(" ")

**val** words **=** spark.sparkContext.parallelize(myCollection, 2)

*# in Python*

myCollection = "Spark The Definitive Guide : Big Data Processing Made Simple"\

.split(" ")

words = spark.sparkContext.parallelize(myCollection, 2)

An additional feature is that you can then name this RDD to show up in the Spark UI according to a given name:

*// in Scala*

words.setName("myWords")

words.name *// myWords*

*# in Python*

words.setName("myWords")

words.name() *# myWords*

**From Data Sources**

Although you can create RDDs from data sources or text files, it’s often preferable to use the Data Source APIs. RDDs do not have a notion of “Data Source APIs” like DataFrames do; they primarily define their dependency structures and lists of partitions. The Data Source API that we saw in [Chapter 9](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch09.html#s2c6---data-sources) is almost always a better way to read in data. That being said, you can also read data as RDDs using sparkContext. For example, let’s read a text file line by line:

spark.sparkContext.textFile("/some/path/withTextFiles")

This creates an RDD for which each record in the RDD represents a line in that text file or files. Alternatively, you can read in data for which each text file should become a single record. The use case here would be where each file is a file that consists of a large JSON object or some document that you will operate on as an individual:

spark.sparkContext.wholeTextFiles("/some/path/withTextFiles")

In this RDD, the name of the file is the first object and the value of the text file is the second string object.

**Manipulating RDDs**

You manipulate RDDs in much the same way that you manipulate DataFrames. As mentioned, the core difference being that you manipulate raw Java or Scala objects instead of Spark types. There is also a dearth of “helper” methods or functions that you can draw upon to simplify calculations. Rather, you must define each filter, map functions, aggregation, and any other manipulation that you want as a function.

To demonstrate some data manipulation, let’s use the simple RDD (words) we created previously to define some more details.

**Transformations**

For the most part, many transformations mirror the functionality that you find in the Structured APIs. Just as you do with DataFrames and Datasets, you specify *transformations* on one RDD to create another. In doing so, we define an RDD as a dependency to another along with some manipulation of the data contained in that RDD.

**distinct**

A distinct method call on an RDD removes duplicates from the RDD:

words.distinct().count()

This gives a result of 10.

**filter**

Filtering is equivalent to creating a SQL-like where clause. You can look through our records in the RDD and see which ones match some predicate function. This function just needs to return a Boolean type to be used as a filter function. The input should be whatever your given row is. In this next example, we filter the RDD to keep only the words that begin with the letter “S”:

*// in Scala*

**def** startsWithS(individual**:String**) **=** {

individual.startsWith("S")

}

*# in Python*

**def** startsWithS(individual):

**return** individual.startswith("S")

Now that we defined the function, let’s filter the data. This should feel quite familiar if you read [Chapter 11](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch11.html#s2c8---datasets) because we simply use a function that operates record by record in the RDD. The function is defined to work on each record in the RDD individually:

*// in Scala*

words.filter(word **=>** startsWithS(word)).collect()

*# in Python*

words.filter(**lambda** word: startsWithS(word)).collect()

This gives a result of *Spark* and *Simple*. We can see, like the Dataset API, that this returns native types. That is because we never coerce our data into type Row, nor do we need to convert the data after collecting it.

**map**

Mapping is again the same operation that you can read about in [Chapter 11](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch11.html#s2c8---datasets). You specify a function that returns the value that you want, given the correct input. You then apply that, record by record. Let’s perform something similar to what we just did. In this example, we’ll map the current word to the word, its starting letter, and whether the word begins with “S.”

Notice in this instance that we define our functions completely inline using the relevant lambda syntax:

*// in Scala*

**val** words2 **=** words.map(word **=>** (word, word(0), word.startsWith("S")))

*# in Python*

words2 = words.map(**lambda** word: (word, word[0], word.startswith("S")))

You can subsequently filter on this by selecting the relevant Boolean value in a new function:

*// in Scala*

words2.filter(record **=>** record.\_3).take(5)

*# in Python*

words2.filter(**lambda** record: record[2]).take(5)

This returns a tuple of “Spark,” “S,” and “true,” as well as “Simple,” “S,” and “True.”

**FLATMAP**

flatMap provides a simple extension of the map function we just looked at. Sometimes, each current row should return multiple rows, instead. For example, you might want to take your set of words and flatMap it into a set of characters. Because each word has multiple characters, you should use flatMap to expand it. flatMap requires that the ouput of the map function be an iterable that can be expanded:

*// in Scala*

words.flatMap(word **=>** word.toSeq).take(5)

*# in Python*

words.flatMap(**lambda** word: list(word)).take(5)

This yields *S*, *P*, *A*, *R*, *K*.

**sort**

To sort an RDD you must use the sortBy method, and just like any other RDD operation, you do this by specifying a function to extract a value from the objects in your RDDs and then sort based on that. For instance, the following example sorts by word length from longest to shortest:

*// in Scala*

words.sortBy(word **=>** word.length() \* -1).take(2)

*# in Python*

words.sortBy(**lambda** word: len(word) \* -1).take(2)

**Random Splits**

We can also randomly split an RDD into an Array of RDDs by using the randomSplit method, which accepts an Array of weights and a random seed:

*// in Scala*

**val** fiftyFiftySplit **=** words.randomSplit(**Array**[**Double**](0.5, 0.5))

*# in Python*

fiftyFiftySplit = words.randomSplit([0.5, 0.5])

This returns an array of RDDs that you can manipulate individually.

**Actions**

Just as we do with DataFrames and Datasets, we specify *actions* to kick off our specified transformations. Actions either collect data to the driver or write to an external data source.

**reduce**

You can use the reduce method to specify a function to “reduce” an RDD of any kind of value to one value. For instance, given a set of numbers, you can reduce this to its sum by specifying a function that takes as input two values and reduces them into one. If you have experience in functional programming, this should not be a new concept:

*// in Scala*

spark.sparkContext.parallelize(1 to 20).reduce(**\_** + **\_**) *// 210*

*# in Python*

spark.sparkContext.parallelize(range(1, 21)).reduce(**lambda** x, y: x + y) *# 210*

You can also use this to get something like the longest word in our set of words that we defined a moment ago. The key is just to define the correct function:

*// in Scala*

**def** wordLengthReducer(leftWord**:String**, rightWord**:String**)**:** **String** = {

**if** (leftWord.length > rightWord.length)

**return** leftWord

**else**

**return** rightWord

}

words.reduce(wordLengthReducer)

*# in Python*

**def** wordLengthReducer(leftWord, rightWord):

**if** len(leftWord) > len(rightWord):

**return** leftWord

**else**:

**return** rightWord

words.reduce(wordLengthReducer)

This reducer is a good example because you can get one of two outputs. Because the reduce operation on the partitions is not deterministic, you can have either “definitive” or “processing” (both of length 10) as the “left” word. This means that sometimes you can end up with one, whereas other times you end up with the other.

**count**

This method is fairly self-explanatory. Using it, you could, for example, count the number of rows in the RDD:

words.count()

**COUNTAPPROX**

Even though the return signature for this type is a bit strange, it’s quite sophisticated. This is an approximation of the count method we just looked at, but it must execute within a timeout (and can return incomplete results if it exceeds the timeout).

The confidence is the probability that the error bounds of the result will contain the true value. That is, if countApprox were called repeatedly with confidence 0.9, we would expect 90% of the results to contain the true count. The confidence must be in the range [0,1], or an exception will be thrown:

**val** confidence **=** 0.95

**val** timeoutMilliseconds **=** 400

words.countApprox(timeoutMilliseconds, confidence)

**COUNTAPPROXDISTINCT**

There are two implementations of this, both based on streamlib’s implementation of “HyperLogLog in Practice: Algorithmic Engineering of a State-of-the-Art Cardinality Estimation Algorithm.”

In the first implementation, the argument we pass into the function is the relative accuracy. Smaller values create counters that require more space. The value must be greater than 0.000017:

words.countApproxDistinct(0.05)

With the other implementation you have a bit more control; you specify the relative accuracy based on two parameters: one for “regular” data and another for a sparse representation.

The two arguments are p and sp where p is precision and sp is sparse precision. The relative accuracy is approximately 1.054 / sqrt(2P). Setting a nonzero (sp > p) can reduce the memory consumption and increase accuracy when the cardinality is small. Both values are integers:

words.countApproxDistinct(4, 10)

**COUNTBYVALUE**

This method counts the number of values in a given RDD. However, it does so by finally loading the result set into the memory of the driver. You should use this method only if the resulting map is expected to be small because the entire thing is loaded into the driver’s memory. Thus, this method makes sense only in a scenario in which either the total number of rows is low or the number of distinct items is low:

words.countByValue()

**COUNTBYVALUEAPPROX**

This does the same thing as the previous function, but it does so as an approximation. This must execute within the specified timeout (first parameter) (and can return incomplete results if it exceeds the timeout).

The confidence is the probability that the error bounds of the result will contain the true value. That is, if countApprox were called repeatedly with confidence 0.9, we would expect 90% of the results to contain the true count. The confidence must be in the range [0,1], or an exception will be thrown:

words.countByValueApprox(1000, 0.95)

**first**

The first method returns the first value in the dataset:

words.first()

**max and min**

max and min return the maximum values and minimum values, respectively:

spark.sparkContext.parallelize(1 to 20).max()

spark.sparkContext.parallelize(1 to 20).min()

**take**

take and its derivative methods take a number of values from your RDD. This works by first scanning one partition and then using the results from that partition to estimate the number of additional partitions needed to satisfy the limit.

There are many variations on this function, such as takeOrdered, takeSample, and top. You can use takeSample to specify a fixed-size random sample from your RDD. You can specify whether this should be done by using withReplacement, the number of values, as well as the random seed. top is effectively the opposite of takeOrdered in that it selects the top values according to the implicit ordering:

words.take(5)

words.takeOrdered(5)

words.top(5)

**val** withReplacement **=** **true**

**val** numberToTake **=** 6

**val** randomSeed **=** 100L

words.takeSample(withReplacement, numberToTake, randomSeed)

**Saving Files**

Saving files means writing to plain-text files. With RDDs, you cannot actually “save” to a data source in the conventional sense. You must iterate over the partitions in order to save the contents of each partition to some external database. This is a low-level approach that reveals the underlying operation that is being performed in the higher-level APIs. Spark will take each partition, and write that out to the destination.

**saveAsTextFile**

To save to a text file, you just specify a path and optionally a compression codec:

words.saveAsTextFile("file:/tmp/bookTitle")

To set a compression codec, we must import the proper codec from Hadoop. You can find these in the org.apache.hadoop.io.compress library:

*// in Scala*

**import** **org.apache.hadoop.io.compress.BZip2Codec**

words.saveAsTextFile("file:/tmp/bookTitleCompressed", classOf[**BZip2Codec**])

**SequenceFiles**

Spark originally grew out of the Hadoop ecosystem, so it has a fairly tight integration with a variety of Hadoop tools. A sequenceFile is a flat file consisting of binary key–value pairs. It is extensively used in MapReduce as input/output formats.

Spark can write to sequenceFiles using the saveAsObjectFile method or by explicitly writing key–value pairs, as described in [Chapter 13](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch13.html#s3c2---advanced-rdds):

words.saveAsObjectFile("/tmp/my/sequenceFilePath")

**Hadoop Files**

There are a variety of different Hadoop file formats to which you can save. These allow you to specify classes, output formats, Hadoop configurations, and compression schemes. (For information on these formats, read *Hadoop: The Definitive Guide* [O’Reilly, 2015].) These formats are largely irrelevant except if you’re working deeply in the Hadoop ecosystem or with some legacy mapReduce jobs.

**Caching**

The same principles apply for caching RDDs as for DataFrames and Datasets. You can either cache or persist an RDD. By default, cache and persist only handle data in memory. We can name it if we use the setName function that we referenced previously in this chapter:

words.cache()

We can specify a storage level as any of the storage levels in the singleton object: org.apache.spark.storage.StorageLevel, which are combinations of memory only; disk only; and separately, off heap.

We can subsequently query for this storage level (we talk about storage levels when we discuss persistence in [Chapter 20](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch20.html#s5c0---stream-processing-fundamentals)):

*// in Scala*

words.getStorageLevel

*# in Python*

words.getStorageLevel()

**Checkpointing**

One feature not available in the DataFrame API is the concept of *checkpointing*. Checkpointing is the act of saving an RDD to disk so that future references to this RDD point to those intermediate partitions on disk rather than recomputing the RDD from its original source. This is similar to caching except that it’s not stored in memory, only disk. This can be helpful when performing iterative computation, similar to the use cases for caching:

spark.sparkContext.setCheckpointDir("/some/path/for/checkpointing")

words.checkpoint()

Now, when we reference this RDD, it will derive from the checkpoint instead of the source data. This can be a helpful optimization.

**Pipe RDDs to System Commands**

The pipe method is probably one of Spark’s more interesting methods. With pipe, you can return an RDD created by piping elements to a forked external process. The resulting RDD is computed by executing the given process once per partition. All elements of each input partition are written to a process’s stdin as lines of input separated by a newline. The resulting partition consists of the process’s stdout output, with each line of stdout resulting in one element of the output partition. A process is invoked even for empty partitions.

The print behavior can be customized by providing two functions.

We can use a simple example and pipe each partition to the command wc. Each row will be passed in as a new line, so if we perform a line count, we will get the number of lines, one per partition:

words.pipe("wc -l").collect()

In this case, we got five lines per partition.

**mapPartitions**

The previous command revealed that Spark operates on a per-partition basis when it comes to actually executing code. You also might have noticed earlier that the return signature of a map function on an RDD is actually MapPartitionsRDD. This is because map is just a row-wise alias for mapPartitions, which makes it possible for you to map an individual partition (represented as an iterator). That’s because physically on the cluster we operate on each partition individually (and not a specific row). A simple example creates the value “1” for every partition in our data, and the sum of the following expression will count the number of partitions we have:

*// in Scala*

words.mapPartitions(part **=>** **Iterator**[**Int**](1)).sum() *// 2*

*# in Python*

words.mapPartitions(**lambda** part: [1]).sum() *# 2*

Naturally, this means that we operate on a per-partition basis and allows us to perform an operation on that *entire* partition. This is valuable for performing something on an entire subdataset of your RDD. You can gather all values of a partition class or group into one partition and then operate on that entire group using arbitrary functions and controls. An example use case of this would be that you could pipe this through some custom machine learning algorithm and train an individual model for that company’s portion of the dataset. A Facebook engineer has an interesting demonstration of their particular implementation of the pipe operator with a similar use case [demonstrated at Spark Summit East 2017](https://spark-summit.org/east-2017/events/experiences-with-sparks-rdd-apis-for-complex-custom-applications/).

Other functions similar to mapPartitions include mapPartitionsWithIndex. With this you specify a function that accepts an index (within the partition) and an iterator that goes through all items within the partition. The partition index is the partition number in your RDD, which identifies where each record in our dataset sits (and potentially allows you to debug). You might use this to test whether your map functions are behaving correctly:

*// in Scala*

**def** indexedFunc(partitionIndex**:Int**, withinPartIterator**:** **Iterator**[**String**]) **=** {

withinPartIterator.toList.map(

value **=>** s"Partition: $partitionIndex => $value").iterator

}

words.mapPartitionsWithIndex(indexedFunc).collect()

*# in Python*

**def** indexedFunc(partitionIndex, withinPartIterator):

**return** ["partition: {} => {}".format(partitionIndex,

x) **for** x **in** withinPartIterator]

words.mapPartitionsWithIndex(indexedFunc).collect()

**foreachPartition**

Although mapPartitions needs a return value to work properly, this next function does not. foreachPartition simply iterates over all the partitions of the data. The difference is that the function has no return value. This makes it great for doing something with each partition like writing it out to a database. In fact, this is how many data source connectors are written. You can create our own text file source if you want by specifying outputs to the temp directory with a random ID:

words.foreachPartition { iter **=>**

**import** **java.io.\_**

**import** **scala.util.Random**

**val** randomFileName **=** **new** **Random**().nextInt()

**val** pw **=** **new** **PrintWriter**(**new** **File**(s"/tmp/random-file-${randomFileName}.txt"))

**while** (iter.hasNext) {

pw.write(iter.next())

}

pw.close()

}

You’ll find these two files if you scan your */tmp* directory.

**glom**

glom is an interesting function that takes every partition in your dataset and converts them to arrays. This can be useful if you’re going to collect the data to the driver and want to have an array for each partition. However, this can cause serious stability issues because if you have large partitions or a large number of partitions, it’s simple to crash the driver.

In the following example, you can see that we get two partitions and each word falls into one partition each:

*// in Scala*

spark.sparkContext.parallelize(**Seq**("Hello", "World"), 2).glom().collect()

*// Array(Array(Hello), Array(World))*

*# in Python*

spark.sparkContext.parallelize(["Hello", "World"], 2).glom().collect()

*# [['Hello'], ['World']]*

**Conclusion**

In this chapter, you saw the basics of the RDD APIs, including single RDD manipulation. [Chapter 13](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch13.html#s3c2---advanced-rdds) touches on more advanced RDD concepts, such as joins and key-value RDDs.