**Chapter 13. Advanced RDDs**

[Chapter 12](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch12.html#s3c1---rdd-basics) explored the basics of single RDD manipulation. You learned how to create RDDs and why you might want to use them. In addition, we discussed map, filter, reduce, and how to create functions to transform single RDD data. This chapter covers the advanced RDD operations and focuses on key–value RDDs, a powerful abstraction for manipulating data. We also touch on some more advanced topics like custom partitioning, a reason you might want to use RDDs in the first place. With a custom partitioning function, you can control exactly how data is laid out on the cluster and manipulate that individual partition accordingly. Before we get there, let’s summarize the key topics we will cover:

* Aggregations and key–value RDDs
* Custom partitioning
* RDD joins

**NOTE**

This set of APIs has been around since, essentially, the beginning of Spark, and there are a *ton* of examples all across the web on this set of APIs. This makes it trivial to search and find examples that will show you how to use these operations.

Let’s use the same dataset we used in the last chapter:

*// 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)

**Key-Value Basics (Key-Value RDDs)**

There are many methods on RDDs that require you to put your data in a key–value format. A hint that this is required is that the method will include <some-operation>ByKey. Whenever you see ByKey in a method name, it means that you can perform this only on a PairRDD type. The easiest way is to just map over your current RDD to a basic key–value structure. This means having two values in each record of your RDD:

*// in Scala*

words.map(word **=>** (word.toLowerCase, 1))

*# in Python*

words.map(**lambda** word: (word.lower(), 1))

**keyBy**

The preceding example demonstrated a simple way to create a key. However, you can also use the keyBy function to achieve the same result by specifying a function that creates the key from your current value. In this case, you are keying by the first letter in the word. Spark then keeps the record as the value for the keyed RDD:

*// in Scala*

**val** keyword **=** words.keyBy(word **=>** word.toLowerCase.toSeq(0).toString)

*# in Python*

keyword = words.keyBy(**lambda** word: word.lower()[0])

**Mapping over Values**

After you have a set of key–value pairs, you can begin manipulating them as such. If we have a tuple, Spark will assume that the first element is the key, and the second is the value. When in this format, you can explicitly choose to map-over the values (and ignore the individual keys). Of course, you could do this manually, but this can help prevent errors when you know that you are just going to modify the values:

*// in Scala*

keyword.mapValues(word **=>** word.toUpperCase).collect()

*# in Python*

keyword.mapValues(**lambda** word: word.upper()).collect()

Here’s the output in Python:

[('s', 'SPARK'),

('t', 'THE'),

('d', 'DEFINITIVE'),

('g', 'GUIDE'),

(':', ':'),

('b', 'BIG'),

('d', 'DATA'),

('p', 'PROCESSING'),

('m', 'MADE'),

('s', 'SIMPLE')]

(The values in Scala are the same but omitted for brevity.)

You can flatMap over the rows, as we saw in [Chapter 12](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch12.html#s3c1---rdd-basics), to expand the number of rows that you have to make it so that each row represents a character. In the following example, we will omit the output, but it would simply be each character as we converted them into arrays:

*// in Scala*

keyword.flatMapValues(word **=>** word.toUpperCase).collect()

*# in Python*

keyword.flatMapValues(**lambda** word: word.upper()).collect()

**Extracting Keys and Values**

When we are in the key–value pair format, we can also extract the specific keys or values by using the following methods:

*// in Scala*

keyword.keys.collect()

keyword.values.collect()

*# in Python*

keyword.keys().collect()

keyword.values().collect()

**lookup**

One interesting task you might want to do with an RDD is look up the result for a particular key. Note that there is *no* enforcement mechanism with respect to there being only one key for each input, so if we lookup “s”, we are going to get both values associated with that—“Spark” and “Simple”:

keyword.lookup("s")

**sampleByKey**

There are two ways to sample an RDD by a set of keys. We can do it via an approximation or exactly. Both operations can do so with or without replacement as well as sampling by a fraction by a given key. This is done via simple random sampling with one pass over the RDD, which produces a sample of size that’s approximately equal to the sum of math.ceil(numItems \* samplingRate) over all key values:

*// in Scala*

**val** distinctChars **=** words.flatMap(word **=>** word.toLowerCase.toSeq).distinct

.collect()

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

**val** sampleMap **=** distinctChars.map(c **=>** (c, **new** **Random**().nextDouble())).toMap

words.map(word **=>** (word.toLowerCase.toSeq(0), word))

.sampleByKey(**true**, sampleMap, 6L)

.collect()

*# in Python*

**import** **random**

distinctChars = words.flatMap(**lambda** word: list(word.lower())).distinct()\

.collect()

sampleMap = dict(map(**lambda** c: (c, random.random()), distinctChars))

words.map(**lambda** word: (word.lower()[0], word))\

.sampleByKey(True, sampleMap, 6).collect()

This method differs from sampleByKey in that you make additional passes over the RDD to create a sample size that’s exactly equal to the sum of math.ceil(numItems \* samplingRate) over all key values with a 99.99% confidence. When sampling without replacement, you need one additional pass over the RDD to guarantee sample size; when sampling with replacement, you need two additional passes:

*// in Scala*

words.map(word **=>** (word.toLowerCase.toSeq(0), word))

.sampleByKeyExact(**true**, sampleMap, 6L).collect()

**Aggregations**

You can perform aggregations on plain RDDs or on PairRDDs, depending on the method that you are using. Let’s use some of our datasets to demonstrate this:

*// in Scala*

**val** chars **=** words.flatMap(word **=>** word.toLowerCase.toSeq)

**val** **KVcharacters** **=** chars.map(letter **=>** (letter, 1))

**def** maxFunc(left**:Int**, right**:Int**) **=** math.max(left, right)

**def** addFunc(left**:Int**, right**:Int**) **=** left + right

**val** nums **=** sc.parallelize(1 to 30, 5)

*# in Python*

chars = words.flatMap(**lambda** word: word.lower())

KVcharacters = chars.map(**lambda** letter: (letter, 1))

**def** maxFunc(left, right):

**return** max(left, right)

**def** addFunc(left, right):

**return** left + right

nums = sc.parallelize(range(1,31), 5)

After you have this, you can do something like countByKey, which counts the items per each key.

**countByKey**

You can count the number of elements for each key, collecting the results to a local Map. You can also do this with an approximation, which makes it possible for you to specify a timeout and confidence when using Scala or Java:

*// in Scala*

**val** timeout **=** 1000L *//milliseconds*

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

**KVcharacters**.countByKey()

**KVcharacters**.countByKeyApprox(timeout, confidence)

*# in Python*

KVcharacters.countByKey()

**Understanding Aggregation Implementations**

There are several ways to create your key–value PairRDDs; however, the implementation is actually quite important for job stability. Let’s compare the two fundamental choices, groupBy and reduce. We’ll do these in the context of a key, but the same basic principles apply to the groupBy and reduce methods.

**GROUPBYKEY**

Looking at the API documentation, you might think groupByKey with a map over each grouping is the best way to sum up the counts for each key:

*// in Scala*

**KVcharacters**.groupByKey().map(row **=>** (row.\_1, row.\_2.reduce(addFunc))).collect()

*# in Python*

KVcharacters.groupByKey().map(**lambda** row: (row[0], reduce(addFunc, row[1])))\

.collect()

*# note this is Python 2, reduce must be imported from functools in Python 3*

However, this is, for the majority of cases, the wrong way to approach the problem. The fundamental issue here is that each executor must hold *all values* for a given key in memory before applying the function to them. Why is this problematic? If you have massive key skew, some partitions might be completely overloaded with a ton of values for a given key, and you will get OutOfMemoryErrors. This obviously doesn’t cause an issue with our current dataset, but it can cause serious problems at scale. This is not guaranteed to happen, but it *can* happen.

There are use cases when groupByKey does make sense. If you have consistent value sizes for each key and know that they will fit in the memory of a given executor, you’re going to be just fine. It’s just good to know exactly what you’re getting yourself into when you do this. There is a preferred approach for additive use cases: reduceByKey.

**REDUCEBYKEY**

Because we are performing a simple count, a much more stable approach is to perform the same flatMap and then just perform a map to map each letter instance to the number one, and then perform a reduceByKey with a summation function in order to collect back the array. This implementation is much more stable because the reduce happens within each partition and doesn’t need to put everything in memory. Additionally, there is no incurred shuffle during this operation; everything happens at each worker individually before performing a final reduce. This greatly enhances the speed at which you can perform the operation as well as the stability of the operation:

**KVcharacters**.reduceByKey(addFunc).collect()

Here’s the result of the operation:

Array((d,4), (p,3), (t,3), (b,1), (h,1), (n,2),

...

(a,4), (i,7), (k,1), (u,1), (o,1), (g,3), (m,2), (c,1))

The reduceByKey method returns an RDD of a group (the key) and sequence of elements that are not guranteed to have an ordering. Therefore this method is completely appropriate when our workload is associative but inappropriate when the order matters.

**Other Aggregation Methods**

There exist a number of advanced aggregation methods. For the most part these are largely implementation details depending on your specific workload. We find it very rare that users come across this sort of workload (or need to perform this kind of operation) in modern-day Spark. There just aren’t that many reasons for using these extremely low-level tools when you can perform much simpler aggregations using the Structured APIs. These functions largely allow you very specific, very low-level control on exactly how a given aggregation is performed on the cluster of machines.

**AGGREGATE**

Another function is aggregate. This function requires a null and start value and then requires you to specify two different functions. The first aggregates within partitions, the second aggregates across partitions. The start value will be used at both aggregation levels:

*// in Scala*

nums.aggregate(0)(maxFunc, addFunc)

*# in Python*

nums.aggregate(0, maxFunc, addFunc)

aggregate does have some performance implications because it performs the final aggregation on the driver. If the results from the executors are too large, they can take down the driver with an OutOfMemoryError. There is another method, treeAggregate that does the same thing as aggregate (at the user level) but does so in a different way. It basically “pushes down” some of the subaggregations (creating a tree from executor to executor) before performing the final aggregation on the driver. Having multiple levels can help you to ensure that the driver does not run out of memory in the process of the aggregation. These tree-based implementations are often to try to improve stability in certain operations:

*// in Scala*

**val** depth **=** 3

nums.treeAggregate(0)(maxFunc, addFunc, depth)

*# in Python*

depth = 3

nums.treeAggregate(0, maxFunc, addFunc, depth)

**AGGREGATEBYKEY**

This function does the same as aggregate but instead of doing it partition by partition, it does it by key. The start value and functions follow the same properties:

*// in Scala*

**KVcharacters**.aggregateByKey(0)(addFunc, maxFunc).collect()

*# in Python*

KVcharacters.aggregateByKey(0, addFunc, maxFunc).collect()

**COMBINEBYKEY**

Instead of specifying an aggregation function, you can specify a combiner. This combiner operates on a given key and merges the values according to some function. It then goes to merge the different outputs of the combiners to give us our result. We can specify the number of output partitions as a custom output partitioner as well:

*// in Scala*

**val** valToCombiner **=** (value**:Int**) **=>** **List**(value)

**val** mergeValuesFunc **=** (vals**:List**[**Int**], valToAppend**:Int**) **=>** valToAppend :: vals

**val** mergeCombinerFunc **=** (vals1**:List**[**Int**], vals2**:List**[**Int**]) **=>** vals1 ::: vals2

*// now we define these as function variables*

**val** outputPartitions **=** 6

**KVcharacters**

.combineByKey(

valToCombiner,

mergeValuesFunc,

mergeCombinerFunc,

outputPartitions)

.collect()

*# in Python*

**def** valToCombiner(value):

**return** [value]

**def** mergeValuesFunc(vals, valToAppend):

vals.append(valToAppend)

**return** vals

**def** mergeCombinerFunc(vals1, vals2):

**return** vals1 + vals2

outputPartitions = 6

KVcharacters\

.combineByKey(

valToCombiner,

mergeValuesFunc,

mergeCombinerFunc,

outputPartitions)\

.collect()

**FOLDBYKEY**

foldByKey merges the values for each key using an associative function and a neutral “zero value,” which can be added to the result an arbitrary number of times, and must not change the result (e.g., 0 for addition, or 1 for multiplication):

*// in Scala*

**KVcharacters**.foldByKey(0)(addFunc).collect()

*# in Python*

KVcharacters.foldByKey(0, addFunc).collect()

**CoGroups**

CoGroups give you the ability to group together up to three key–value RDDs together in Scala and two in Python. This joins the given values by key. This is effectively just a group-based join on an RDD. When doing this, you can also specify a number of output partitions or a custom partitioning function to control exactly how this data is distributed across the cluster (we talk about partitioning functions later on in this chapter):

*// in Scala*

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

**val** distinctChars **=** words.flatMap(word **=>** word.toLowerCase.toSeq).distinct

**val** charRDD **=** distinctChars.map(c **=>** (c, **new** **Random**().nextDouble()))

**val** charRDD2 **=** distinctChars.map(c **=>** (c, **new** **Random**().nextDouble()))

**val** charRDD3 **=** distinctChars.map(c **=>** (c, **new** **Random**().nextDouble()))

charRDD.cogroup(charRDD2, charRDD3).take(5)

*# in Python*

**import** **random**

distinctChars = words.flatMap(**lambda** word: word.lower()).distinct()

charRDD = distinctChars.map(**lambda** c: (c, random.random()))

charRDD2 = distinctChars.map(**lambda** c: (c, random.random()))

charRDD.cogroup(charRDD2).take(5)

The result is a group with our key on one side, and all of the relevant values on the other side.

**Joins**

RDDs have much the same joins as we saw in the Structured API, although RDDs are much more involved for you. They all follow the same basic format: the two RDDs we would like to join, and, optionally, either the number of output partitions or the customer partition function to which they should output. We’ll talk about partitioning functions later on in this chapter.

**Inner Join**

We’ll demonstrate an inner join now. Notice how we are setting the number of output partitions we would like to see:

*// in Scala*

**val** keyedChars **=** distinctChars.map(c **=>** (c, **new** **Random**().nextDouble()))

**val** outputPartitions **=** 10

**KVcharacters**.join(keyedChars).count()

**KVcharacters**.join(keyedChars, outputPartitions).count()

*# in Python*

keyedChars = distinctChars.map(**lambda** c: (c, random.random()))

outputPartitions = 10

KVcharacters.join(keyedChars).count()

KVcharacters.join(keyedChars, outputPartitions).count()

We won’t provide an example for the other joins, but they all follow the same basic format. You can learn about the following join types at the conceptual level in [Chapter 8](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch08.html#s2c5---joins):

* fullOuterJoin
* leftOuterJoin
* rightOuterJoin
* cartesian (This, again, is very dangerous! It does not accept a join key and can have a massive output.)

**zips**

The final type of join isn’t really a join at all, but it does combine two RDDs, so it’s worth labeling it as a join. zip allows you to “zip” together two RDDs, assuming that they have the same length. This creates a PairRDD. The two RDDs must have the same number of partitions as well as the same number of elements:

*// in Scala*

**val** numRange **=** sc.parallelize(0 to 9, 2)

words.zip(numRange).collect()

*# in Python*

numRange = sc.parallelize(range(10), 2)

words.zip(numRange).collect()

This gives us the following result, an array of keys zipped to the values:

[('Spark', 0),

('The', 1),

('Definitive', 2),

('Guide', 3),

(':', 4),

('Big', 5),

('Data', 6),

('Processing', 7),

('Made', 8),

('Simple', 9)]

**Controlling Partitions**

With RDDs, you have control over how data is exactly physically distributed across the cluster. Some of these methods are basically the same from what we have in the Structured APIs but the key addition (that does not exist in the Structured APIs) is the ability to specify a partitioning function (formally a custom Partitioner, which we discuss later when we look at basic methods).

**coalesce**

coalesce effectively collapses partitions on the same worker in order to avoid a shuffle of the data when repartitioning. For instance, our words RDD is currently two partitions, we can collapse that to one partition by using coalesce without bringing about a shuffle of the data:

*// in Scala*

words.coalesce(1).getNumPartitions *// 1*

*# in Python*

words.coalesce(1).getNumPartitions() *# 1*

**repartition**

The repartition operation allows you to repartition your data up or down but performs a shuffle across nodes in the process. Increasing the number of partitions can increase the level of parallelism when operating in map- and filter-type operations:

words.repartition(10) *// gives us 10 partitions*

**repartitionAndSortWithinPartitions**

This operation gives you the ability to repartition as well as specify the ordering of each one of those output partitions. We’ll omit the example because the documentation for it is good, but both the partitioning and the key comparisons can be specified by the user.

**Custom Partitioning**

This ability is one of the primary reasons you’d want to use RDDs. Custom partitioners are not available in the Structured APIs because they don’t really have a logical counterpart. They’re a low-level, implementation detail that can have a significant effect on whether your jobs run successfully. The canonical example to motivate custom partition for this operation is PageRank whereby we seek to control the layout of the data on the cluster and avoid shuffles. In our shopping dataset, this might mean partitioning by each customer ID (we’ll get to this example in a moment).

In short, the sole goal of custom partitioning is to even out the distribution of your data across the cluster so that you can work around problems like data skew.

If you’re going to use custom partitioners, you should drop down to RDDs from the Structured APIs, apply your custom partitioner, and then convert it back to a DataFrame or Dataset. This way, you get the best of both worlds, only dropping down to custom partitioning when you need to.

To perform custom partitioning you need to implement your own class that extends Partitioner. You need to do this only when you have lots of domain knowledge about your problem space—if you’re just looking to partition on a value or even a set of values (columns), it’s worth just doing it in the DataFrame API.

Let’s dive into an example:

*// in Scala*

**val** df **=** spark.read.option("header", "true").option("inferSchema", "true")

.csv("/data/retail-data/all/")

**val** rdd **=** df.coalesce(10).rdd

*# in Python*

df = spark.read.option("header", "true").option("inferSchema", "true")\

.csv("/data/retail-data/all/")

rdd = df.coalesce(10).rdd

df.printSchema()

Spark has two built-in Partitioners that you can leverage off in the RDD API, a HashPartitioner for discrete values and a RangePartitioner. These two work for discrete values and continuous values, respectively. Spark’s Structured APIs will already use these, although we can use the same thing in RDDs:

*// in Scala*

**import** **org.apache.spark.HashPartitioner**

rdd.map(r **=>** r(6)).take(5).foreach(println)

**val** keyedRDD **=** rdd.keyBy(row **=>** row(6).asInstanceOf[**Int**].toDouble)

keyedRDD.partitionBy(**new** **HashPartitioner**(10)).take(10)

Although the hash and range partitioners are useful, they’re fairly rudimentary. At times, you will need to perform some very low-level partitioning because you’re working with very large data and large *key skew*. Key skew simply means that some keys have many, many more values than other keys. You want to break these keys as much as possible to improve parallelism and prevent OutOfMemoryErrors during the course of execution.

One instance might be that you need to partition more keys if and only if the key matches a certain format. For instance, we might know that there are two customers in your dataset that always crash your analysis and we need to break them up further than other customer IDs. In fact, these two are so skewed that they need to be operated on alone, whereas all of the others can be lumped into large groups. This is obviously a bit of a caricatured example, but you might see similar situations in your data, as well:

*// in Scala*

**import** **org.apache.spark.Partitioner**

**class** **DomainPartitioner** **extends** **Partitioner** {

**def** numPartitions **=** 3

**def** getPartition(key**:** **Any**)**:** **Int** = {

**val** customerId **=** key.asInstanceOf[**Double**].toInt

**if** (customerId == 17850.0 || customerId == 12583.0) {

**return** 0

} **else** {

**return** **new** java.util.**Random**().nextInt(2) + 1

}

}

}

keyedRDD

.partitionBy(**new** **DomainPartitioner**).map(**\_**.\_1).glom().map(**\_**.toSet.toSeq.length)

.take(5)

After you run this, you will see the count of results in each partition. The second two numbers will vary, because we’re distributing them randomly (as you will see when we do the same in Python) but the same principles apply:

*# in Python*

**def** partitionFunc(key):

**import** **random**

**if** key == 17850 **or** key == 12583:

**return** 0

**else**:

**return** random.randint(1,2)

keyedRDD = rdd.keyBy(**lambda** row: row[6])

keyedRDD\

.partitionBy(3, partitionFunc)\

.map(**lambda** x: x[0])\

.glom()\

.map(**lambda** x: len(set(x)))\

.take(5)

This custom key distribution logic is available only at the RDD level. Of course, this is a simple example, but it does show the power of using arbitrary logic to distribute the data around the cluster in a physical manner.

**Custom Serialization**

The last advanced topic that is worth talking about is the issue of *Kryo serialization*. Any object that you hope to parallelize (or function) must be serializable:

*// in Scala*

**class** **SomeClass** **extends** **Serializable** {

**var** someValue **=** 0

**def** setSomeValue(i**:Int**) **=** {

someValue **=** i

**this**

}

}

sc.parallelize(1 to 10).map(num **=>** **new** **SomeClass**().setSomeValue(num))

The default serialization can be quite slow. Spark can use the Kryo library (version 2) to serialize objects more quickly. Kryo is significantly faster and more compact than Java serialization (often as much as 10x), but does not support all serializable types and requires you to register the classes you’ll use in the program in advance for best performance.

You can use Kryo by initializing your job with a SparkConf and setting the value of "spark.serializer" to "org.apache.spark.serializer.KryoSerializer" (we discuss this in the next part of the book). This setting configures the serializer used for shuffling data between worker nodes and serializing RDDs to disk. The only reason Kryo is not the default is because of the custom registration requirement, but we recommend trying it in any network-intensive application. Since Spark 2.0.0, we internally use Kryo serializer when shuffling RDDs with simple types, arrays of simple types, or string type.

Spark automatically includes Kryo serializers for the many commonly used core Scala classes covered in the AllScalaRegistrar from the Twitter chill library.

To register your own custom classes with Kryo, use the registerKryoClasses method:

*// in Scala*

**val** conf **=** **new** **SparkConf**().setMaster(...).setAppName(...)

conf.registerKryoClasses(**Array**(classOf[**MyClass1**], classOf[**MyClass2**]))

**val** sc **=** **new** **SparkContext**(conf)

**Conclusion**

In this chapter we discussed many of the more advanced topics regarding RDDs. Of particular note was the section on custom partitioning, which allows you very specific functions to layout your data. In [Chapter 14](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch14.html#s3c3---distributed-variables), we discuss another of Spark’s low-level tools: distributed variables.