3
Spark RDD

Resilient Distributed Datasets (**RDDs**) are the basic building block of a Spark application. An RDD represents a read-only collection of objects distributed across multiple machines. Spark can distribute a collection of records using an RDD and process them in parallel on different machines.

In this chapter, we shall learn about the following:

- What is an RDD?
- How do you create RDDs?
- Different operations available to work on RDDs
- Important types of RDD
- Caching an RDD
- Partitions of an RDD
- Drawbacks of using RDDs

The code examples in this chapter are written in Python and Scala only. If you wish to go through the Java and R APIs, you can visit the Spark documentation page at `https://spark.apache.org/`.

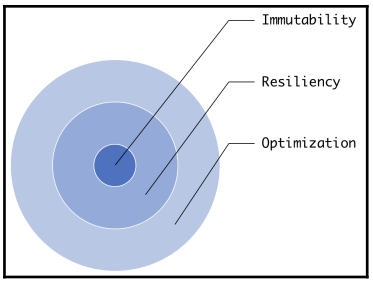
Copyright © 2019. Packt Publishing, Limited. All rights reserved.

Mehrotra, Shrey, and Akash Grade. Apache Spark Quick Start Guide : Quickly Learn the Art of Writing Efficient Big Data Applications with Apache Spark, Packt Publishing, Limited, 2019. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/etown-ebooks/detail.action?docID=5675596.
Created from etown-ebooks on 2025-01-12 23:17:59.

What is an RDD?

RDD is at the heart of every Spark application. Let's understand the meaning of each word in more detail:

- **Resilient**: If we look at the meaning of *resilient* in the dictionary, we can see that it means to be: *able to recover quickly from difficult conditions.* Spark RDD has the ability to recreate itself if something goes wrong. You must be wondering, *why does it need to recreate itself?* Remember how HDFS and other data stores achieve fault tolerance? Yes, these systems maintain a replica of the data on multiple machines to recover in case of failure. But, as discussed in `Chapter 1`, *Introduction to Apache Spark*, Spark is not a data store; Spark is an execution engine. It reads the data from source systems, transforms it, and loads it into the target system. If something goes wrong while performing any of the previous steps, we will lose the data. To provide the fault tolerance while processing, an RDD is made resilient: it can recompute itself. Each RDD maintains some information about its parent RDD and how it was created from its parent. This introduces us to the concept of **Lineage**. The information about maintaining the parent and the operation is known as lineage. Lineage can only be achieved if your data is **immutable**. What do I mean by that? If you lose the current state of an object and you are sure that previous state will never change, then you can always go back and use its past state with the same operations, and you will always recover the current state of the object. This is exactly what happens in the case of RDDs. If you are finding this difficult, don't worry! It will become clear when we look at how RDDs are created.
  Immutability also brings another advantage: o*ptimization*. If you know something will not change, you always have the opportunity to optimize it. If you pay close attention, all of these concepts are connected, as the following diagram illustrates:



RDD

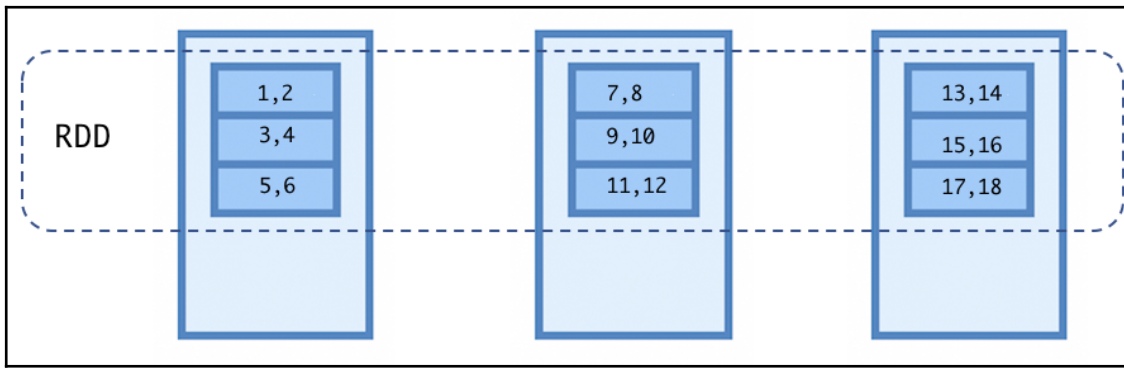
Copyright © 2019. Packt Publishing, Limited. All rights reserved.

Mehrotra, Shrey, and Akash Grade. Apache Spark Quick Start Guide : Quickly Learn the Art of Writing Efficient Big Data Applications with Apache Spark, Packt Publishing, Limited, 2019. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/etown-ebooks/detail.action?docID=5675596.
Created from etown-ebooks on 2025-01-12 23:17:59.

- **Distributed**: As mentioned in the following bullet point, a dataset is nothing but a collection of objects. An RDD can distribute its dataset across a set of machines, and each of these machines will be responsible for processing its *partition* of data. If you come from a Hadoop MapReduce background, you can imagine partitions as the input splits for the map phase.
- **Dataset**: A dataset is just a collection of objects. These objects can be a Scala, Java, or Python complex object; numbers; strings; rows of a database; and more.

Every Spark program boils down to an RDD. A Spark program written in Spark SQL, DataFrame, or dataset gets converted to an RDD at the time of execution.

The following diagram illustrates an **RDD** of numbers (**1** to **18**) having nine partitions on a cluster of three nodes:



RDD

Resilient metadata

As we have discussed, apart from partitions, an RDD also stores some metadata within it. This metadata helps Spark to recompute an RDD partition in the case of failure and also provides optimizations while performing operations.

The metadata includes the following:

- A list of parent RDD dependencies
- A function to compute a partition from the list of parent RDDs
- The preferred location for the partitions
- The partitioning information, in case of pair RDDs

Copyright © 2019. Packt Publishing, Limited. All rights reserved.

Mehrotra, Shrey, and Akash Grade. Apache Spark Quick Start Guide : Quickly Learn the Art of Writing Efficient Big Data Applications with Apache Spark, Packt Publishing, Limited, 2019. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/etown-ebooks/detail.action?docID=5675596.
Created from etown-ebooks on 2025-01-12 23:17:59.

Right then, enough theory! Let's create a simple program and understand the concepts in more detail in the next section.

Programming using RDDs

An RDD can be created in four ways:

- **Parallelize a collection**: This is one of the easiest ways to create an RDD. You can use the existing collection from your programs, such as `List`, `Array`, or `Set`, as well as others, and ask Spark to distribute that collection across the cluster to process it in parallel. A collection can be distributed with the help of `parallelize()`, as shown here:

```
#Python
numberRDD = spark.sparkContext.parallelize(range(1,10))
numberRDD.collect()

Out[4]: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

  The following code performs the same operation in Scala:

```
//scala
val numberRDD = spark.sparkContext.parallelize(1 to 10)
numberRDD.collect()

res4: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

- **From an external dataset**: Though parallelizing a collection is the easiest way to create an RDD, it is not the recommended way for the large datasets. Large datasets are generally stored on filesystems such as HDFS, and we know that Spark is built to process big data. Therefore, Spark provides a number of APIs to read data from the external datasets. One of the methods for reading external data is the `textFile()`. This method accepts a filename and creates an RDD, where each element of the RDD is the line of the input file.

  In the following example, we first initialize a variable with the file path and then use the `filePath` variable as an argument of `textFile()` method:

```
//Scala
val filePath = "/FileStore/tables/sampleFile.log"
val logRDD = spark.sparkContext.textFile(filePath)
logRDD.collect()
```

Copyright © 2019. Packt Publishing, Limited. All rights reserved.

Mehrotra, Shrey, and Akash Grade. Apache Spark Quick Start Guide : Quickly Learn the Art of Writing Efficient Big Data Applications with Apache Spark, Packt Publishing, Limited, 2019. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/etown-ebooks/detail.action?docID=5675596.
Created from etown-ebooks on 2025-01-12 23:17:59.

```
res6: Array[String] = Array(2018-03-19 17:10:26 - myApp - DEBUG -
debug message 1, 2018-03-19 17:10:27 - myApp - INFO - info message
1, 2018-03-19 17:10:28 - myApp - WARNING - warn message 1,
2018-03-19 17:10:29 - myApp - ERROR - error message 1, 2018-03-19
17:10:32 - myApp - CRITICAL - critical message with some error 1,
2018-03-19 17:10:33 - myApp - INFO - info message 2, 2018-03-19
17:10:37 - myApp - WARNING - warn message, 2018-03-19 17:10:41 -
myApp - ERROR - error message 2, 2018-03-19 17:10:41 - myApp -
ERROR - error message 3)
```

If your data is present in multiple files, you can make use of `wholeTextFiles()` instead of using the `textFile()` method. The argument to `wholeTextFiles()` is the directory name that contains all the files. Each element will be represented as a key value pair, where the key will be the file name and the value will be the whole content of that file. This is useful in scenarios where you have lots of small files and want to process each file separately.

JSON and XML file are common inputs of `wholeTextFiles()` as you can parse each file separately using a parser library.

- **From another RDD**: As discussed in the first section, RDDs are immutable in nature. They cannot be modified, but we can transform an RDD to another RDD with the help of the methods provided by Spark. We shall discuss these methods in more detail in this chapter. The following example uses `filter()` to transform our `numberRDD` to `evenNumberRDD` in Python. Similarly, it also uses `filter()` to create `oddNumberRDD` in Scala:

```
#Python
evenNumberRDD = numberRDD.filter(lambda num : num%2 == 0 )
evenNumberRDD.collect()

Out[10]: [2, 4, 6, 8]
```

The following code performs the same operation in Scala:

```
//Scala
val oddNumberRDD = numberRDD.filter( num => num%2 != 0 )
oddNumberRDD.collect()

res8: Array[Int] = Array(1, 3, 5, 7, 9)
```

Copyright © 2019. Packt Publishing, Limited. All rights reserved.

Mehrotra, Shrey, and Akash Grade. Apache Spark Quick Start Guide : Quickly Learn the Art of Writing Efficient Big Data Applications with Apache Spark, Packt Publishing, Limited, 2019. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/etown-ebooks/detail.action?docID=5675596.
Created from etown-ebooks on 2025-01-12 23:17:59.

- **From a DataFrame or dataset**: You must be thinking, why would we ever create an RDD from a DataFrame? After all, a DataFrame is an abstraction on top of an RDD. Well, you're right! Because of this, it is advisable to use DataFrames or a dataset over an RDD, because a DataFrame brings performance benefits.

You might need to convert an RDD from a DataFrame in some scenarios where the following applies:

- The data is highly unstructured
- The data is reduced to a manageable size after heavy computations, such as joins or aggregations, and you want more control over the physical distribution of data using custom partitioning
- You have some code written in a different programming language or legacy RDD code

Let's create a DataFrame and convert it into an RDD:

```
#Python
rangeDf = spark.range(1,5)
rangeRDD = rangeDf.rdd
rangeRDD.collect()

Out[15]: [Row(id=1), Row(id=2), Row(id=3), Row(id=4)]
```

In the preceding code, we first created a `rangeDf`DataFrame with an `id` column (the default column name) using Spark's `range()` method, which created `4` rows, from `1` to `4`. We then use the `rdd` method to convert it into `rangeRDD`.

The `range(N)` method creates values from `0` to `N-1`.

As we have now got a basic understanding of how to create RDDs, let's write a simple program that reads a log file and returns only the number of messages with log levels of `ERROR` and `INFO`:

```
$ cat sampleFile.log
2018-03-19 17:10:26 - myApp - DEBUG - debug message 1
2018-03-19 17:10:27 - myApp - INFO - info message 1
2018-03-19 17:10:28 - myApp - WARNING - warn message 1
2018-03-19 17:10:29 - myApp - ERROR - error message 1
2018-03-19 17:10:32 - myApp - CRITICAL - critical message with some error 1
2018-03-19 17:10:33 - myApp - INFO - info message 2
2018-03-19 17:10:37 - myApp - WARNING - warn message
```


Copyright © 2019. Packt Publishing, Limited. All rights reserved.

Mehrotra, Shrey, and Akash Grade. Apache Spark Quick Start Guide : Quickly Learn the Art of Writing Efficient Big Data Applications with Apache Spark, Packt Publishing, Limited, 2019. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/etown-ebooks/detail.action?docID=5675596.
Created from etown-ebooks on 2025-01-12 23:17:59.

```
2018-03-19 17:10:41 - myApp - ERROR - error message 2
2018-03-19 17:10:41 - myApp - ERROR - error message 3
```

The preceding code shows the content of the `sampleFile.log` files. Each line in `sampleFile.log` represents a log with its log level.

The next code snippets calculates the number of `ERROR` and `INFO` messages in the log file using the Python API:

```
#Python
filePath = "/FileStore/tables/sampleFile.log"

logRDD = spark.sparkContext.textFile(filePath)

resultRDD = logRDD.filter(lambda line : line.split(" - ")[2] in
['INFO','ERROR'])\
                  .map(lambda line : (line.split(" - ")[2], 1))\
                  .reduceByKey(lambda x, y : x + y)

resultRDD.collect()


Out[27]: [('INFO', 2), ('ERROR', 3)]
```

The following code performs the same operation in Scala:

```
//Scala
val filePath = "/FileStore/table/sampleFile.log"

val logRDD = spark.sparkContext.textFile(filePath)

val resultRDD = logRDD.filter(line =>
Array("INFO","ERROR").contains(line.split(" -")(2)))
 .map(line => (line.split(" - ")(2), 1))
 .reduceByKey( _ + _ )

resultRDD.collect()


res12: Array[(String, Int)] = Array((ERROR,3), (INFO,2))
```

Copyright © 2019. Packt Publishing, Limited. All rights reserved.

Mehrotra, Shrey, and Akash Grade. Apache Spark Quick Start Guide : Quickly Learn the Art of Writing Efficient Big Data Applications with Apache Spark, Packt Publishing, Limited, 2019. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/etown-ebooks/detail.action?docID=5675596.
Created from etown-ebooks on 2025-01-12 23:17:59.

In the preceding two examples, we first created a `filePath` variable that contained the path to our log file. We then made use of the `textFile()` method to create our base RDD, that is `logRDD`. Under the hood, Spark adds this operation into its DAG. At the time of execution, Spark will read our `sampleFile.log` and distribute it to multiple executors. In the next line, we make use of `filter()` to get only those lines that have `"INFO"` and `"ERROR"` as the log level. The `filter()` method accepts a function as input and returns a Boolean. We also pipe the output of `filter` to a `map()` object, and now the problem is reduced to the word-count problem. At this point, `map()` will only receive the filtered lines and assign `1` to each record. We aggregate the records based on the log level using `reduceByKey()`, which adds all the values for each log level. We finally collect our result using the `collect()` method. This is the point where Spark actually starts executing the DAG.

Transformations and actions

We have discussed some basic operations for creating and manipulating RDDs. Now it is time to categorize them into two main categories:

- Transformations
- Actions

Transformation

As the name suggests, transformations help us in transforming existing RDDs. As an output, they always create a new RDD that gets computed lazily. In the previous examples, we have discussed many transformations, such as `map()`, `filter()`, and `reduceByKey()`.

Transformations are of two types:

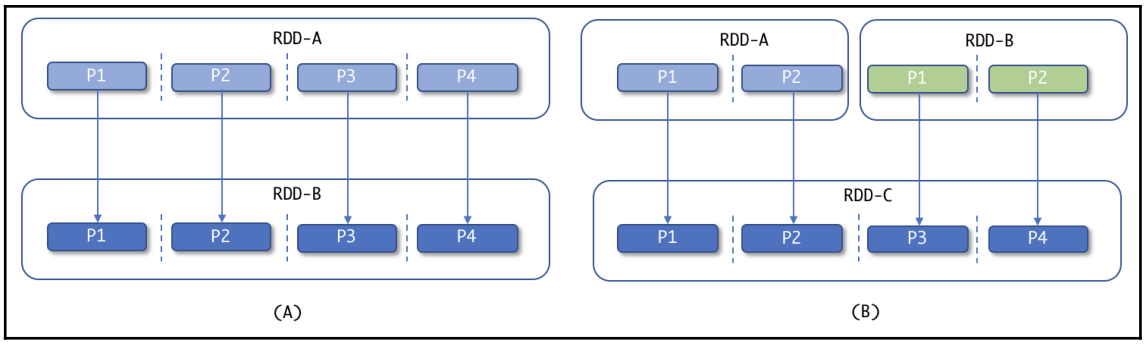
- Narrow transformations
- Wide transformations

Copyright © 2019. Packt Publishing, Limited. All rights reserved.

Mehrotra, Shrey, and Akash Grade. Apache Spark Quick Start Guide : Quickly Learn the Art of Writing Efficient Big Data Applications with Apache Spark, Packt Publishing, Limited, 2019. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/etown-ebooks/detail.action?docID=5675596.
Created from etown-ebooks on 2025-01-12 23:17:59.

Narrow transformations

Narrow transformations transform data without any shuffle involved. These transformations transform the data on a per-partition basis; that is to say, each element of the output RDD can be computed without involving any elements from different partitions. This leads to an important point: The new RDD will always have the same number of partitions as its parent RDDs, and that's why they are easy to recompute in the case of failure. Let's understand this with the following example:



Narrow transformations

So, we have an **RDD-A** and we perform a narrow transformation, such as `map()` or `filter()`, and we get a new **RDD-B** with the same number of partitions as **RDD-A**. In part **(B)**, we have two, **RDD-A** and **RDD-B**, and we perform another type of narrow transformation such as `union()`, and we get a new **RDD-C** with the number of partitions equal to the sum of partitions of its parent RDDs (**A** and **B**). Let's look at some examples of narrow transformations.

map()

This applies a given function to each element of an RDD and returns a new RDD with the same number of elements. For example, in the following code, numbers from `1` to `10` are multiplied by the number `2`:

```
#Python
spark.sparkContext.parallelize(range(1,11)).map(lambda x : x * 2).collect()
```

Copyright © 2019. Packt Publishing, Limited. All rights reserved.

Mehrotra, Shrey, and Akash Grade. Apache Spark Quick Start Guide : Quickly Learn the Art of Writing Efficient Big Data Applications with Apache Spark, Packt Publishing, Limited, 2019. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/etown-ebooks/detail.action?docID=5675596.
Created from etown-ebooks on 2025-01-12 23:17:59.

The following code performs the same operation in Scala:

```
//Scala
spark.sparkContext.parallelize(1 to 10).map(_ * 2).collect()
```

flatMap()

This applies a given function that returns an iterator to each element of an RDD and returns a new RDD with more elements. In some cases, you might need multiple elements from a single element. For example, in the following code, an RDD containing lines is converted into another RDD containing words:

```
#Python
spark.sparkContext.parallelize(["It's fun to learn Spark","This is a
flatMap example using Python"]).flatMap(lambda x : x.split(" ")).collect()
```

The following code performs the same operation in Scala:

```
//Scala
spark.sparkContext.parallelize(Array("It's fun to learn Spark","This is a
flatMap example using Python")).flatMap(x => x.split(" ")).collect()
```

filter()

The `filter()` transformation applies a function that filters out the elements that do not pass the condition criteria, as shown in the following code. For example, if we need numbers greater than `5`, we can pass this condition to the `filter()` transformation. Let's create an RDD of numbers `1` to `10` and filter out numbers that are greater than `5`:

```
#Python
spark.sparkContext.parallelize(range(1,11)).filter(lambda x : x >
5).collect()
```

The following code performs the same operation in Scala:

```
//Scala
spark.sparkContext.parallelize(1 to 10).filter(_ > 5).collect()
```

Any function that returns a Boolean value can be used used to filter out the elements.

Copyright © 2019. Packt Publishing, Limited. All rights reserved.

Mehrotra, Shrey, and Akash Grade. Apache Spark Quick Start Guide : Quickly Learn the Art of Writing Efficient Big Data Applications with Apache Spark, Packt Publishing, Limited, 2019. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/etown-ebooks/detail.action?docID=5675596.
Created from etown-ebooks on 2025-01-12 23:17:59.

union()

The `union()` transformation takes another RDD as an input and produces a new RDD containing elements from both the RDDs, as shown in the following code. Let's create two RDDs: one with numbers `1` to `5` and another with numbers `5` to `10`, and then concatenate them together to get a new RDD with the numbers `1` to `10`:

```
#Python
firstRDD = spark.sparkContext.parallelize(range(1,6))
secordRDD = spark.sparkContext.parallelize(range(5,11))
firstRDD.union(secordRDD).collect()
```

The following code performs the same operation in Scala:

```
//scala
val firstRDD = spark.sparkContext.parallelize(1 to 5)
val secordRDD = spark.sparkContext.parallelize(5 to 10)
firstRDD.union(secordRDD).collect()
```

The `union()` transformation does not remove duplicates. If you are coming from a SQL background, `union()` performs the same operation as `Union All` in SQL.

mapPartitions()

The `mapPartitions()` transformation is similar to `map()`. It also allows users to manipulate elements of an RDD, but it provides more control at a per-partition basis. It applies a function that accepts an iterator as an argument and returns an iterator as the output. If you have done some shell scripting and you are aware of **AWK** programming, then you can correlate that with `mapPartitions` transformation to understand it better. A typical AWK example looks something like `BEGIN { #Begin block } { #middle block } END { #end Block }`. The `Begin` block executes only once before reading the file content, the `middle` block executes for each line in the input file, and the `end` block also executes only once at the end of the file. Similarly, if you want some operations to be performed at the beginning or end of processing all elements one by one, you can make use of the `mapPartitions()` transformation. In the following code, we are multiplying each element by `2`, but this time with `mapPartitions()`:

```
#Python
spark.sparkContext.parallelize(range(1,11), 2).mapPartitions(lambda
iterOfElements : [e*2 for e in iterOfElements]).collect()
```

Copyright © 2019. Packt Publishing, Limited. All rights reserved.

Mehrotra, Shrey, and Akash Grade. Apache Spark Quick Start Guide : Quickly Learn the Art of Writing Efficient Big Data Applications with Apache Spark, Packt Publishing, Limited, 2019. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/etown-ebooks/detail.action?docID=5675596.
Created from etown-ebooks on 2025-01-12 23:17:59.

The following code performs the same operation in Scala:

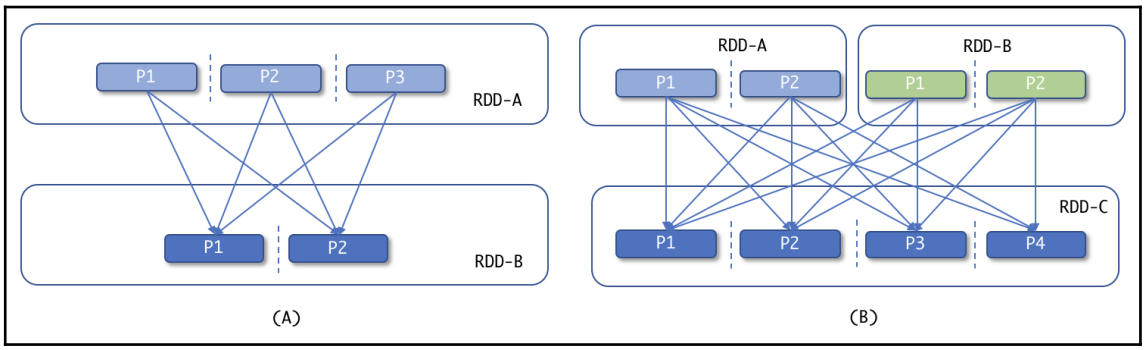
```
//scala
spark.sparkContext.parallelize(1 to 10, 2).mapPartitions(iterOfElements =>
for (e <- iterOfElements) yield e*2 ).collect()
```

One example where you might use `mapPartitions()` is when you need to open a database connection at the beginning of each partition.

If you want to create an object only once and want that object to be used during computation in each partition, you can make use of `broadcast` variables.

Wide transformations

Wide transformations involve a shuffle of the data between the partitions. The `groupByKey()`, `reduceByKey()`, `join()`, `distinct()`, and `intersect()` are some examples of wide transformations. In the case of these transformations, the result will be computed using data from multiple partitions and thus requires a shuffle. Wide transformations are similar to the shuffle-and-sort phase of MapReduce. Let's understand the concept with the help of the following example:



Wide transformations

We have an **RDD-A** and we perform a wide transformation such as `groupByKey()` and we get a new **RDD-B** with fewer partitions. **RDD-B** will have data grouped by each key in the dataset. In part **(B)**, we have two RDDs: **RDD-A**, and **RDD-B** and we perform another type of wide transformation such as `join()` or `intersection()` and get a new **RDD-C**. The following are some examples of wide transformations.

Copyright © 2019. Packt Publishing, Limited. All rights reserved.

Mehrotra, Shrey, and Akash Grade. Apache Spark Quick Start Guide : Quickly Learn the Art of Writing Efficient Big Data Applications with Apache Spark, Packt Publishing, Limited, 2019. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/etown-ebooks/detail.action?docID=5675596.
Created from etown-ebooks on 2025-01-12 23:17:59.

distinct()

The `distinct()` transformation removes duplicate elements and returns a new RDD with unique elements as shown. Let's create an RDD with some duplicate elements (`1,2,3,4`) and use `distinct()` to get an RDD with unique numbers:

```
#Python
spark.sparkContext.parallelize([1,1,2,2,3,3,4,4]).distinct().collect()
```

The following code performs the same operation in Scala:

```
//scala
spark.sparkContext.parallelize(Array(1,1,2,2,3,3,4,4)).distinct().collect()
```

sortBy()

We can sort an RDD with the help of `sortBy()` transformation. It accepts a function that can be used to sort the RDD elements. In the following example, we sort our RDD in descending order using the second element of the tuple:

```
#Python
spark.sparkContext.parallelize([('Rahul', 4),('Aman', 2),('Shrey',
6),('Akash', 1)]).sortBy(lambda x : -x[1]).collect()
```

The following code performs the same operation in Scala:

```
//scala
spark.sparkContext.parallelize(Array(("Rahul", 4),("Aman", 2),("Shrey",
6),("Akash", 1))).sortBy( _._2 * -1 ).collect()
```

The previous code will result in this:

```
[('Shrey', 6), ('Rahul', 4), ('Aman', 2), ('Akash', 1)]
```

intersection()

The `intersection()` transformation allows us to find common elements between two RDDs. Like `union()` transformation, `intersection()` is also a set operation between two RDDs, but involves a shuffle. The following examples show how to find common elements between two RDDs using `intersection()`:

```
#Python
firstRDD = spark.sparkContext.parallelize(range(1,6))
secordRDD = spark.sparkContext.parallelize(range(5,11))
firstRDD.intersection(secordRDD).collect()
```

Copyright © 2019. Packt Publishing, Limited. All rights reserved.

Mehrotra, Shrey, and Akash Grade. Apache Spark Quick Start Guide : Quickly Learn the Art of Writing Efficient Big Data Applications with Apache Spark, Packt Publishing, Limited, 2019. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/etown-ebooks/detail.action?docID=5675596.
Created from etown-ebooks on 2025-01-12 23:17:59.

The following code performs the same operation in Scala:

```
//Scala
val firstRDD = spark.sparkContext.parallelize(1 to 5)
val secordRDD = spark.sparkContext.parallelize(5 to 10)
firstRDD.intersection(secordRDD).collect()
```

The previous code gives a result of `5`.

subtract()

You can use `subtract()` transformation to remove the content of one RDD using another RDD. Let's create two RDDs: The first one has numbers from `1` to `10` and the second one has elements from `6` to `10`. If we use `subtract()`, we get a new RDD with numbers `1` to `5`:

```
#Python
firstRDD = spark.sparkContext.parallelize(range(1,11))
secordRDD = spark.sparkContext.parallelize(range(6,11))
firstRDD.subtract(secordRDD).collect()
```

The following code performs the same operation in Scala:

```
//scala
val firstRDD = spark.sparkContext.parallelize(1 to 10)
val secordRDD = spark.sparkContext.parallelize(6 to 10)
firstRDD.subtract(secordRDD).collect()
```

In the previous example, we have two RDDs: `firstRDD` contains elements from `1` to `10` and `secondRDD` contains elements `6` to `10`. After applying the `subtract()` transformation, we get a new RDD containing elements from `1` to `5`.

cartesian()

The `cartesian()` transformation can join elements of one RDD with all the elements of another RDD and results in the cartesian product of two. In the following examples, `firstRDD` has elements [`0`, `1`, `2`] and `secondRDD` has elements `['A','B','C']`. We use `cartesian()` to get the cartesian product of two RDDs:

```
#Python
firstRDD = spark.sparkContext.parallelize(range(3))
secordRDD = spark.sparkContext.parallelize(['A','B','C'])
firstRDD.cartesian(secordRDD).collect()
```

Copyright © 2019. Packt Publishing, Limited. All rights reserved.

Mehrotra, Shrey, and Akash Grade. Apache Spark Quick Start Guide : Quickly Learn the Art of Writing Efficient Big Data Applications with Apache Spark, Packt Publishing, Limited, 2019. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/etown-ebooks/detail.action?docID=5675596.
Created from etown-ebooks on 2025-01-12 23:17:59.

The following code performs the same operation in Scala:

```
//scala
val firstRDD = spark.sparkContext.parallelize(0 to 2)
val secordRDD = spark.sparkContext.parallelize(Array("A","B","C"))
firstRDD.cartesian(secordRDD).collect()
```

Here is the output from the previous example:

```
//Scala
Array[(Int, String)] = Array((0,A), (0,B), (0,C), (1,A), (1,B), (1,C),
(2,A), (2,B), (2,C))
```

Remember these operations involve a shuffle, and therefore require lots of computing resources such as memory, disk, and network bandwidth.

`textFile()` and `wholeTextFiles()` are also considered transformations, as they create a new RDD from external data.

Action

You would have noticed that in every example we used, the `collect()` method to get the output. To get the final result back to the driver, Spark provides another type of operation known as *actions*. At the time of transformations, Spark chains these operations and constructs a DAG, but nothing gets executed. Once an action is performed on an RDD, it forces the evaluation of all the transformations required to compute that RDD.

Actions do not create a new RDD. They are used for the following:

- Returning final results to the driver
- Writing final result to an external storage
- Performing some operation on each element of that RDD (for example, `foreach()`)

Let's discuss some of the basic actions.

Copyright © 2019. Packt Publishing, Limited. All rights reserved.

Mehrotra, Shrey, and Akash Grade. Apache Spark Quick Start Guide : Quickly Learn the Art of Writing Efficient Big Data Applications with Apache Spark, Packt Publishing, Limited, 2019. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/etown-ebooks/detail.action?docID=5675596.
Created from etown-ebooks on 2025-01-12 23:17:59.

collect()

The `collect()` action returns all the elements of an RDD to the driver program. You should only use `collect()` if you are sure about the size of your final output. If the size of the final output is huge, then your driver program might crash while receiving the data from the executors. The use of `collect()` is not advised in production. The following example collects all the elements of an RDD containing numbers from `0` to `9`:

```
#Python
spark.sparkContext.parallelize(range(10)).collect()

Out[26]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

count()

Use `count()` to count the number of elements in the RDD. The following Scala code counts the number of an RDD and returns `10` as output:

```
//scala
spark.sparkContext.parallelize(1 to 10).count()

res17: Long = 10
```

take()

The `take()` action returns *N* number of elements from an RDD. The following code returns the first two elements from an RDD containing the numbers `0` to `9`:

```
#Python
spark.sparkContext.parallelize(range(10)).take(2)

Out[27]: [0, 1]
```

top()

The `top()` action returns the top *N* elements from the RDD. The following code returns the top `2` elements from an RDD:

```
#Python
spark.sparkContext.parallelize(range(10)).top(2)

Out[28]: [9, 8]
```

Copyright © 2019. Packt Publishing, Limited. All rights reserved.

Mehrotra, Shrey, and Akash Grade. Apache Spark Quick Start Guide : Quickly Learn the Art of Writing Efficient Big Data Applications with Apache Spark, Packt Publishing, Limited, 2019. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/etown-ebooks/detail.action?docID=5675596.
Created from etown-ebooks on 2025-01-12 23:17:59.

takeOrdered()

If you want to get *N* element based on an ordering, you can use a `takeOrdered()` action. You can also make use of `sortBy()` transformation, followed by a `take()` action. Both approaches trigger a data shuffle. In the following example, we take out `3` elements from the RDD, containing numbers from `0` to `9`, by providing our own sorting criteria:

```
#Python
spark.sparkContext.parallelize(range(10)).takeOrdered(3, key = lambda x: -
x)

Out[3]: [9, 8, 7]
```

Here, we took the first `3` elements in decreasing order.

first()

The `first()` action returns the first element of the RDD. The following example returns the first element of the RDD:

```
#Python
spark.sparkContext.parallelize(range(10)).first()

Out[4]: 0
```

countByValue()

The `countByValue()` action can be used to find out the occurrence of each element in the RDD. The following is the Scala code that returns a `Map` of key-value pair. In the output, `Map`, the key is the RDD element, and the value is the number of occurrences of that element in the RDD:

```
//Scala
spark.sparkContext.parallelize(Array("A","A","B","C")).countByValue()

res0: scala.collection.Map[String,Long] = Map(A -> 2, B -> 1, C -> 1)
```

Copyright © 2019. Packt Publishing, Limited. All rights reserved.

Mehrotra, Shrey, and Akash Grade. Apache Spark Quick Start Guide : Quickly Learn the Art of Writing Efficient Big Data Applications with Apache Spark, Packt Publishing, Limited, 2019. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/etown-ebooks/detail.action?docID=5675596.
Created from etown-ebooks on 2025-01-12 23:17:59.

reduce()

The `reduce()` action combines the RDD elements in parallel and gives aggregated results as output. In the following example, we calculate the sum of the first `10` natural numbers:

```
//Scala
spark.sparkContext.parallelize(1 to 10).reduce( _ + _ )

res1: Int = 55
```

saveAsTextFile()

To save the results to an external data store, we can make use of `saveAsTextFile()` to save your result in a directory. You can also specify a compression codec to store your data in compressed form. Let's write our number RDD to a file:

```
#Python
spark.sparkContext.parallelize(range(10)).saveAsTextFile('/FileStore/tables
/result')
```

In the preceding example, we provide a directory as an argument, and Spark writes data inside this directory in multiple files, along with the success file (`_success`).

If an existing directory is provided as an argument to, `saveAsTextFile()` action, then the job will fail with the `FileAlreadyExistsException` exception. This behavior is important because we might rewrite a directory accidentally that holds data from a heavy job.

foreach()

The `foreach()` function applies a function to each element of the RDD. The following example concatenates the string `Mr.` to each element using `foreach()`:

```
//Scala
spark.sparkContext.parallelize(Array("Smith","John","Brown","Dave")).foreac
h{ x => println("Mr. "+x) }
```

If you run the previous example in local mode, you will see the output. But, in the case of cluster mode, you won't be able to see the results, because `foreach()` performs the given function inside the executors and does not return any data to the driver.

Copyright © 2019. Packt Publishing, Limited. All rights reserved.

Mehrotra, Shrey, and Akash Grade. Apache Spark Quick Start Guide : Quickly Learn the Art of Writing Efficient Big Data Applications with Apache Spark, Packt Publishing, Limited, 2019. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/etown-ebooks/detail.action?docID=5675596.
Created from etown-ebooks on 2025-01-12 23:17:59.

This is mainly used to work with accumulators. We shall see this in more detail in `Chapter 5`, *Spark Architecture and Application Execution Flow*.

You can find more transformations and actions at `https://spark.apache.org/docs/2.3.0/rdd-programming-guide.html#transformations`.

Types of RDDs

RDDs can be categorized in multiple categories. Some of the examples include the following:

Hadoop RDD	Shuffled RDD	Pair RDD
Mapped RDD	Union RDD	JSON RDD
Filtered RDD	Double RDD	Vertex RDD

We will not discuss all of them in this chapter, as it is outside the scope of this chapter. But we will discuss one of the important types of RDD: **pair RDDs**.

Pair RDDs

A pair RDD is a special type of RDD that processes data in the form of key-value pairs. Pair RDD is very useful because it enables basic functionalities such as `join` and `aggregations`. Spark provides some special operations on these RDDs in an optimized way. If we recall the examples where we calculated the number of `INFO` and `ERROR` messages in `sampleFile.log` using `reduceByKey()`, we can clearly see the importance of the pair RDD.

One of the ways to create a pair RDD is to parallelize a collection that contains elements in the form of `Tuple`. Let's look at some of the transformations provided by a pair RDD.

groupByKey()

Elements having the same key can be grouped together with the help of a `groupByKey()` transformation. The following example aggregates data for each key:

```
#Python
pairRDD = spark.sparkContext.parallelize([(1, 5),(1, 10),(2, 4),(3, 1),(2, 6)])
result = pairRDD.groupByKey().collect()
for pair in result:
```


Copyright © 2019. Packt Publishing, Limited. All rights reserved.

Mehrotra, Shrey, and Akash Grade. Apache Spark Quick Start Guide : Quickly Learn the Art of Writing Efficient Big Data Applications with Apache Spark, Packt Publishing, Limited, 2019. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/etown-ebooks/detail.action?docID=5675596.
Created from etown-ebooks on 2025-01-12 23:17:59.

```
  print 'key -',pair[0],', value -', list(pair[1])
```

```
Output:
key - 1 , value - [5, 10]
key - 2 , value - [4, 6]
key - 3 , value - [1]
```

The following code performs the same operation in Scala:

```
//Scala
val pairRDD = spark.sparkContext.parallelize(Array((1, 5),(1, 10),(2,
4),(3, 1),(2, 6)))
val result = pairRDD.groupByKey().collect()
result.foreach {
  pair => println("key - "+pair._1+", value -"+pair._2.toList)
}
```

```
Output:
key - 1, value -List(5, 10)
key - 2, value -List(4, 6)
key - 3, value -List(1)
```

The `groupByKey()` transformation is a wide transformation that shuffles data between executors based on the key. An important point here is to note that `groupByKey()` does not aggregate data, it only groups is based on the key. The `groupByKey()` transformation should be used with the caution. If you understand your data really well, then `groupByKey()` can bring some advantages in some scenarios. For example, let's assume you have a key-value data, where the key is the country code and value is the transaction amount, and your data is highly skewed based on the fact that more than 90% of your customers are based in the USA. In this case, if you use `groupByKey()` to group your data, then you might face some issues because Spark will shuffle all the data and try to send records with the USA to a single machine. This might result in a failure. There are some techniques such as **salted keys** to avoid such scenarios.

Despite this drawback, `groupByKey` can be very useful in some scenarios. If you know your data is not skewed and you want to compute multiple aggregations such as `max`, `min`, and `average` using the same underlying data, then you can first group the elements using `groupByKey()` and persist it.

Copyright © 2019. Packt Publishing, Limited. All rights reserved.

Mehrotra, Shrey, and Akash Grade. Apache Spark Quick Start Guide : Quickly Learn the Art of Writing Efficient Big Data Applications with Apache Spark, Packt Publishing, Limited, 2019. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/etown-ebooks/detail.action?docID=5675596.
Created from etown-ebooks on 2025-01-12 23:17:59.

reduceByKey()

A `reduceByKey()` transformation is available on Pair RDD. It allows aggregation of data by minimizing the data shuffle and performs operations on each key in parallel.
A `reduceByKey()` transformation first performs the local aggregation within the executor and then shuffles the aggregated data between each node. In the following example, we calculate the sum for each key using `reduceByKey`:

```
#Python
pairRDD = spark.sparkContext.parallelize([(1, 5),(1, 10),(2, 4),(3, 1),(2,
6)])
pairRDD.reduceByKey(lambda x,y : x+y).collect()

Output:
[(1, 15), (2, 10), (3, 1)]
```

The following code performs the same operation in Scala:

```
//Scala
val pairRDD = spark.sparkContext.parallelize(Array((1, 5),(1, 10),(2,
4),(3, 1),(2, 6)))
pairRDD.reduceByKey(_+_).collect()

Output:
Array[(Int, Int)] = Array((1,15), (2,10), (3,1))
```

A `reduceByKey()` transformation can only be used for associative aggregations, for example: *(A+B) + C = A + (B+C)*.

sortByKey()

The `sortByKey()` can be used to sort the pair RDD based on keys. In the following example, we first create an RDD by parallelizing a list of tuples and then sort it by the first element of the tuple:

```
#Python
pairRDD = spark.sparkContext.parallelize([(1, 5),(1, 10),(2, 4),(3, 1),(2,
6)])
pairRDD.sortByKey().collect()

Output:
[(1, 5), (1, 10), (2, 4), (2, 6), (3, 1)]
```


Copyright © 2019. Packt Publishing, Limited. All rights reserved.

Mehrotra, Shrey, and Akash Grade. Apache Spark Quick Start Guide : Quickly Learn the Art of Writing Efficient Big Data Applications with Apache Spark, Packt Publishing, Limited, 2019. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/etown-ebooks/detail.action?docID=5675596.
Created from etown-ebooks on 2025-01-12 23:17:59.

By default, `sortByKey()` sorts elements in ascending order, but you can change the sorting order by passing your custom ordering. For example, `sortByKey(keyfunc =lambda k: -k)` will sort the RDD in descending order.

join()

The `join()` transformation will join two pair RDDs based on their keys. The following example joins data based on the country and returns only the matching records:

```
//Scala
val salesRDD = spark.sparkContext.parallelize(Array(("US",20),("IND",
30),("UK",10)))
val revenueRDD = spark.sparkContext.parallelize(Array(("US",200),("IND",
300)))

salesRDD.join(revenueRDD).collect()

Output:
Array[(String, (Int, Int))] = Array((US,(20,200)), (IND,(30,300)))
```

There are some more transformations available on pair RDD such as `aggregateByKey()`, `cogroup()`, `leftOuterJoin()`, `rightOuterJoin()`, `subtractByKey()`, and more. Some of the special actions include `countByKey()`, `collectAsMap()`, and `lookup()`.

Caching and checkpointing

Caching and checkpointing are some of the important features of Spark. These operations can improve the performance of your Spark jobs significantly.

Caching

Caching data into memory is one of the main features of Spark. You can cache large datasets in-memory or on-disk depending upon your cluster hardware. You can choose to cache your data in two scenarios:

- Use the same RDD multiple times
- Avoid reoccupation of an RDD that involves heavy computation, such as `join()` and `groupByKey()`

Copyright © 2019. Packt Publishing, Limited. All rights reserved.

Mehrotra, Shrey, and Akash Grade. Apache Spark Quick Start Guide : Quickly Learn the Art of Writing Efficient Big Data Applications with Apache Spark, Packt Publishing, Limited, 2019. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/etown-ebooks/detail.action?docID=5675596.
Created from etown-ebooks on 2025-01-12 23:17:59.

If you want to run multiple actions of an RDD, then it will be a good idea to cache it into the memory so that recompilation of this RDD can be avoided. For example, the following code first takes out a few elements from the RDD and then returns the count of the elements:

```
//Scala
val baseRDD = spark.sparkContext.parallelize(1 to 10)
baseRDD.take(2)
baseRDD.count()
```

The following code makes use of `cache()` to make the application efficient:

```
//Scala
val baseRDD = spark.sparkContext.parallelize(1 to 10)
baseRDD.cache() //Caching baseRDD
baseRDD.take(2)
baseRDD.count()
```

Spark will compute `baseRDD` twice to perform `take()` and `count()` actions. We cache our `baseRDD` and then run the actions. This computes the RDD only once and performs the action on top of cached data. In this example, there might not be much difference in the performance, as here we are dealing with very small datasets. But you can imagine the bottleneck in the case of big data.

Spark does not cache the data immediately as soon as we write the `cache()` operation. But it makes a note of this operation, and once it encounters the first action, it will compute the RDD and cache it based on the caching level.

The following table lists multiple data persistence levels provided by Spark:

Level	Definition
MEMORY_ONLY	Stores data in memory as unserialized Java objects
MEMORY_ONLY_SER	Stores data in memory but as serialized Java objects
MEMORY_AND_DISK	Unserialized Java objects in memory and remaining serialized data on disk
MEMORY_AND_DISK_SER	Serialized Java objects in memory plus remaining serialized data on disk
DISK_ONLY	Stores data on disk
OFF_HEAP	Stores serialized RDD off-heap in Techyon (Spark's in-memory storage)

You can replicate the cached data on two nodes by writing _2 at the end of the persisting level.

Copyright © 2019. Packt Publishing, Limited. All rights reserved.

Mehrotra, Shrey, and Akash Grade. Apache Spark Quick Start Guide : Quickly Learn the Art of Writing Efficient Big Data Applications with Apache Spark, Packt Publishing, Limited, 2019. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/etown-ebooks/detail.action?docID=5675596.
Created from etown-ebooks on 2025-01-12 23:17:59.

One important point to note here is that in the case of the `MEMORY_ONLY` caching level, if some of the data doesn't fit into the memory, the remaining data is not stored inside the disk by default. The remaining partitions are recomputed at the time of execution. Cache is not a transformation nor an action.

It is recommended to unpersist your cached RDDs once you have finished with that RDD. You can call `unpersist()`, which removes the data from memory.

Checkpointing

The life cycle of the cached RDD will end when the Spark session ends. If you have computed an RDD and you want it to use in another Spark program without recomputing it, then you can make use of the `checkpoint()` operation. This allows storing the RDD content on the disk, which can be used for the later operations. Let's discuss this with the help of an example:

```
#Python
baseRDD = spark.sparkContext.parallelize(['A','B','C'])
spark.sparkContext.setCheckpointDir("/FileStore/tables/checkpointing")
baseRDD.checkpoint()
```

We first create a `baseRDD` and set a checkpointing directory using `setCheckpointDir()` method. Finally, we store the content of `baseRDD` using `checkpoint()`.

Understanding partitions

Data partitioning plays a really important role in distributed computing, as it defines the degree of parallelism for the applications. Understating and defining partitions in the right way can significantly improve the performance of Spark jobs. There are two ways to control the degree of parallelism for RDD operations:

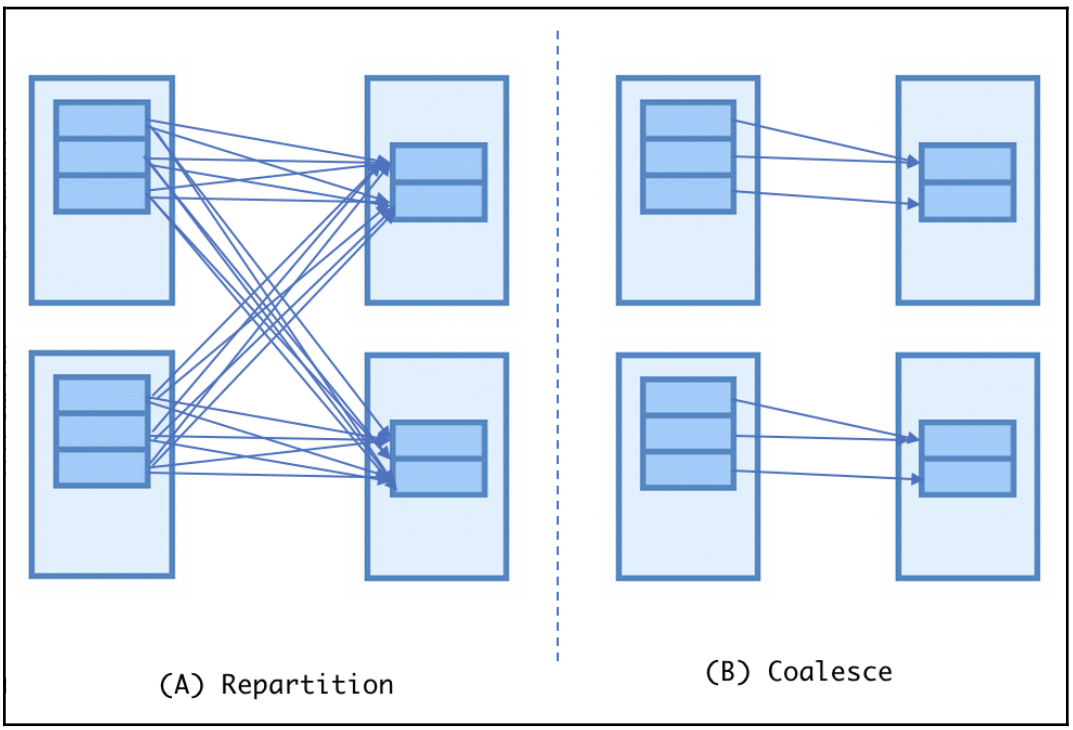
- `repartition()` and `coalesce()`
- `partitionBy()`

Copyright © 2019. Packt Publishing, Limited. All rights reserved.

Mehrotra, Shrey, and Akash Grade. Apache Spark Quick Start Guide : Quickly Learn the Art of Writing Efficient Big Data Applications with Apache Spark, Packt Publishing, Limited, 2019. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/etown-ebooks/detail.action?docID=5675596.
Created from etown-ebooks on 2025-01-12 23:17:59.

repartition() versus coalesce()

Partitions of an existing RDD can be changed using `repartition()` or `coalesce()`. These operations can redistribute the RDD based on the number of partitions provided.
The `repartition()` can be used to increase or decrease the number of partitions, but it involves heavy data shuffling across the cluster. On the other hand, `coalesce()` can be used only to decrease the number of partitions. In most of the cases, `coalesce()` does not trigger a shuffle. The `coalesce()` can be used soon after heavy filtering to optimize the execution time. It is important to notice that `coalesce()` does not always avoid shuffling. If the number of partitions provided is much smaller than the number of available nodes in the cluster then data will be shuffled across some node, but `coalesce()` will still give a better performance than `repartition()`. The following diagram shows the difference between `repartition()` and `coalesce()`:



Repartition and Coalesce

The `repartition()` is not that bad after all. In some cases, when your job is not using all the available slots, you can repartition your data to run it faster.

Copyright © 2019. Packt Publishing, Limited. All rights reserved.

Mehrotra, Shrey, and Akash Grade. Apache Spark Quick Start Guide : Quickly Learn the Art of Writing Efficient Big Data Applications with Apache Spark, Packt Publishing, Limited, 2019. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/etown-ebooks/detail.action?docID=5675596.
Created from etown-ebooks on 2025-01-12 23:17:59.

partitionBy()

Any operation that shuffles the data accepts an additional parameter, that is, degrees of parallelism. This allows users to provide the number of partitions for the produced RDD. The following example shows how you can change the number of partitions of the new RDD by passing an additional parameter:

```
//Scala
val baseRDD = spark.sparkContext.parallelize(Array(("US",20),("IND",
30),("UK",10)), 3)
println(baseRDD.getNumPartitions)

Output:
3
```

The following code changes the number of partitions of the new RDD:

```
//Scala
val groupedRDD = baseRDD.groupByKey(2)
println(groupedRDD.getNumPartitions)

Output:
2
```

The `baseRDD` has `3` partitions. We have passed an additional parameter to `groupByKey` which will tell Spark to produce `groupedRDD` with `2` partitions.

Spark also provides `partitionBy()` operation, which can be used to control the number of partitions. A partitioning function can be passed as an argument to `partitionBy()` to redistribute the data of an RDD. This is quite useful in some operations, such as `join()`. Let's understand this with the help of an example:

```
//Scala
import org.apache.spark.HashPartitioner

val baseRDD = spark.sparkContext.parallelize(Array(("US",20),("IND",
30),("UK",10)), 3)
baseRDD.partitionBy(new HashPartitioner(2)).persist()
```

This shows the usage of `partitionBy()`. We have passed a `HashPartitioner()` that will redistribute the data based on the key values and create two partitions of `baseRDD`. Spark can take advantage of this information and minimize the data shuffle during the `join()` transformation.

Copyright © 2019. Packt Publishing, Limited. All rights reserved.

Mehrotra, Shrey, and Akash Grade. Apache Spark Quick Start Guide : Quickly Learn the Art of Writing Efficient Big Data Applications with Apache Spark, Packt Publishing, Limited, 2019. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/etown-ebooks/detail.action?docID=5675596.
Created from etown-ebooks on 2025-01-12 23:17:59.

It is advisable to persist the RDD after repartitioning if the RDD is going to be used frequently.

Drawbacks of using RDDs

An RDD is a compile-time type-safe. That means, in the case of Scala and Java, if an operation is performed on the RDD that is not applicable to the underlying data type, then Spark will give a compile time error. This can avoid failures in production.

There are some drawbacks of using RDDs though:

- RDD code can sometimes be very opaque. Developers might struggle to find out what exactly the code is trying to compute.
- RDDs cannot be optimized by Spark, as Spark cannot look inside the lambda functions and optimize the operations. In some cases, where a `filter()` is piped after a wide transformation, Spark will never perform the filter first before the wide transformation, such as `reduceByKey()` or `groupByKey()`.
- RDDs are slower on non-JVM languages such as Python and R. In the case of these languages, a Python/R virtual machine is created alongside JVM. There is always a data transfer involved between these VMs, which can significantly increase the execution time.

Summary

In this chapter, we first learned about the basic idea of an RDD. We then looked at how we can create RDDs using different approaches, such as creating an RDD from an existing RDD, from an external data store, from parallelizing a collection, and from a DataFrame and datasets. We also looked at the different types of transformations and actions available on RDDs. Then, the different types of RDDs were discussed, especially the pair RDD. We also discussed the benefits of caching and checkpointing in Spark applications, and then we learned about the partitions in more detail, and how we can make use of features like partitioning, to optimize our Spark jobs.

In the end, we also discussed some of the drawbacks of using RDDs. In the next chapter, we'll discuss the DataFrame and dataset APIs and see how they can overcome these challenges.

Copyright © 2019. Packt Publishing, Limited. All rights reserved.

Mehrotra, Shrey, and Akash Grade. Apache Spark Quick Start Guide : Quickly Learn the Art of Writing Efficient Big Data Applications with Apache Spark, Packt Publishing, Limited, 2019. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/etown-ebooks/detail.action?docID=5675596.
Created from etown-ebooks on 2025-01-12 23:17:59.