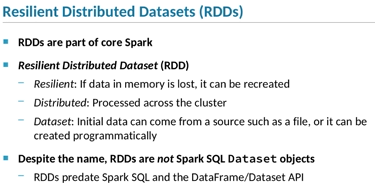
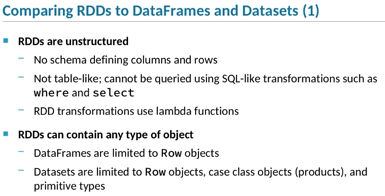
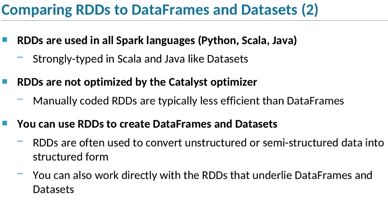
Resilient Distributed Datasets

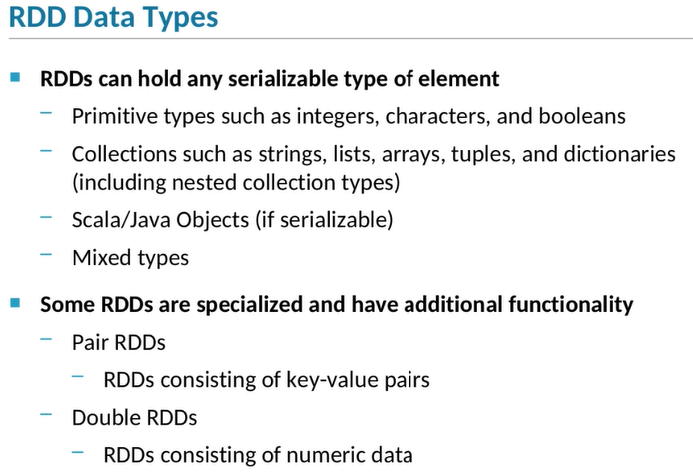


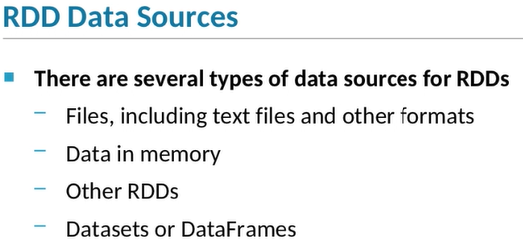


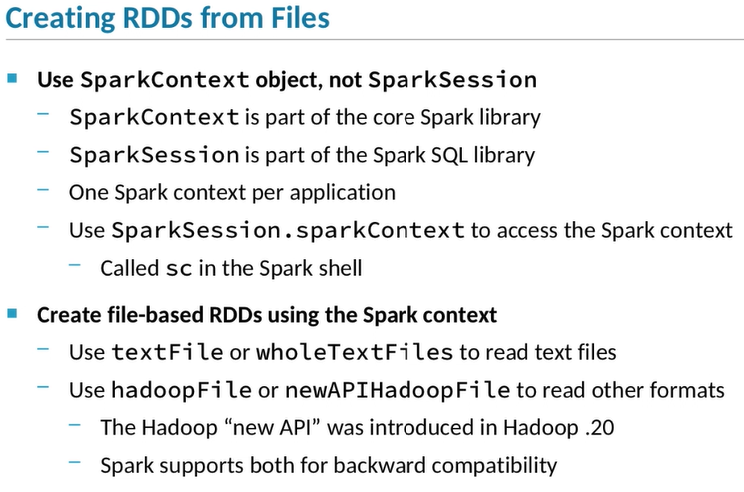


1. This chapter deals with *Resilient Distributed Datasets*, or RDDs.
2. RDDs are part of core Spark and are the basic unit of data in Spark.
3. They represent a collection of data.
4. **RDDs underlie Datasets and DataFrames in Spark SQL.**
5. What does Resilient Distributed Dataset mean?
6. RDDs are resilient because if data in memory is lost—for instance, if an executor crashes while performing a task on an RDD—
7. Spark can recreate the data using the RDD’s lineage.
8. RDDs are also distributed.
9. This means that the data represented by an RDD is stored and processed on nodes throughout the cluster.
10. This architecture provides Spark’s ability to execute tasks in an application in parallel.
11. RDDs are an abstraction representing a set of data, which can come from a variety of sources, such as text files, binary files, and Hive warehouse files.
12. They can also use in-memory data created programmatically.
13. The fact that the word “dataset” is part of the term RDD can be a bit confusing.
14. RDDs are not actually based on the Spark SQL Dataset class.
15. RDDs have been part of Spark from the very beginning, before the Spark SQL library was developed, so the name was defined well before the Dataset API.
16. As you know, Datasets and DataFrames are the main entry point for developing Spark applications.
17. RDDs are the fundamental structure underlying the Dataset and DataFrame implementation.
18. Both represent a set of data and support a variety of operations on that data.
19. So what’s the difference?
20. DataFrames and Datasets represent structured data in a table-like form, which allows the data to be queried using functions like where and select.
21. This sort of querying is quite similar to SQL, and in fact you can perform SQL queries on DataFrames and Datasets.
22. RDDs, on the other hand, are unstructured.
23. In many cases, this means that the underlying data is unstructured, like text.
24. But even in cases where the data itself has some structure—for instance, when the underlying data source is a JSON file—
25. RDDs do not apply a schema to that structure.
26. RDDs don’t include any concept of tables, rows, and columns, and therefore can’t be queried like a table.
27. Another difference is that RDD elements can be objects of any type.
28. DataFrames, by definition, only contain Row objects.
29. Datasets are limited to Rowobjects, product objects, and primitive types like ints and strings.
30. RDDs are used in all the languages Spark supports, including Python, Scala, and Java.
31. When you use RDDs with strongly-typed languages such as Java and Scala, RDDs maintain that strong typing, enforcing type safety.
32. In this sense, RDDs are similar to Datasets, but not to DataFrames.
33. One of the key advantages of the Spark SQL library—the Catalyst optimizer—is not applicable to RDDs.
34. This is because Catalyst needs to understand exactly which transformation tasks are part of a query.
35. RDD transformations are based on functions that the developer creates and passes,
36. and therefore Catalyst can’t determine exactly how the data will be transformed before the task is executed.
37. Although RDDs represent a different concept than DataFrames and Datasets, the two can be used in complementary ways in an application.
38. One important use of RDDs with Spark SQL is to convert unstructured or semi-structured data to the sort of structured data that DataFrames and Datasets were designed for.
39. You can use RDD operations to transform the data into the correct form, and then create a DataFrame or Dataset based on that RDD.
40. Similarly, you can also work directly with the RDDs that underlie DataFrames and Datasets.
41. This is particularly useful when you want to perform an operation that doesn’t fit into the query paradigm.

(RDD) Sources

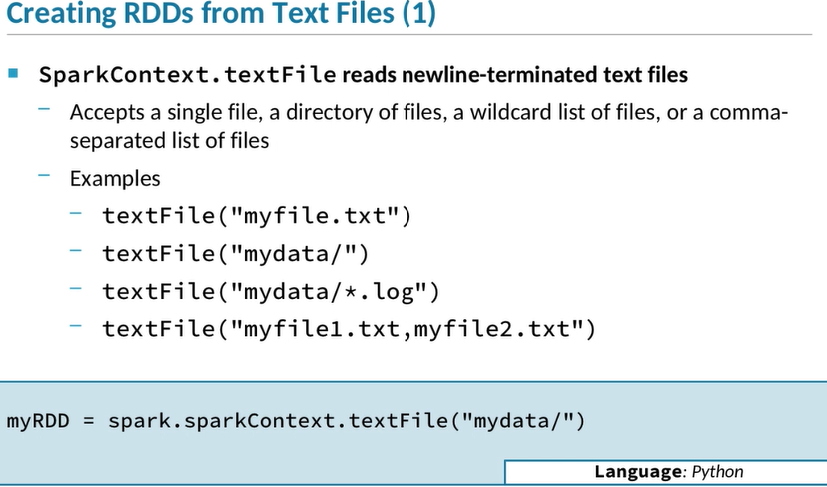


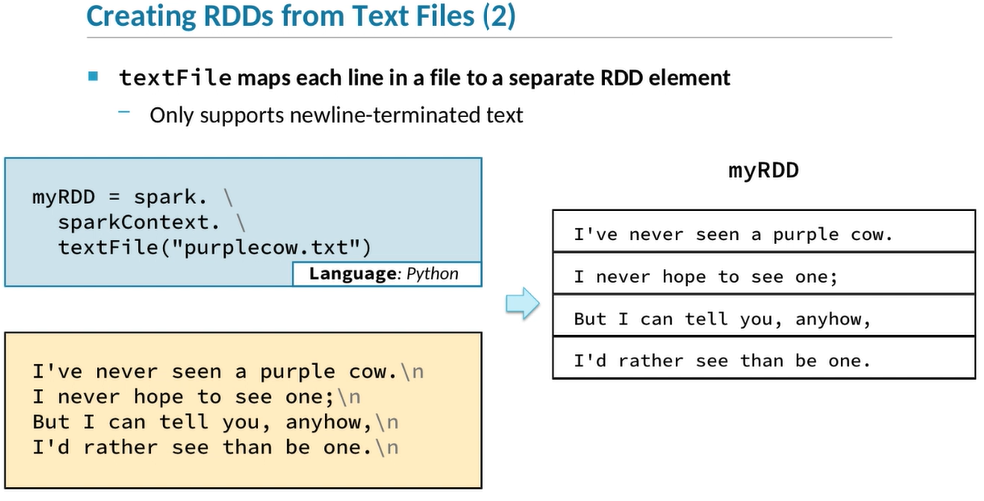


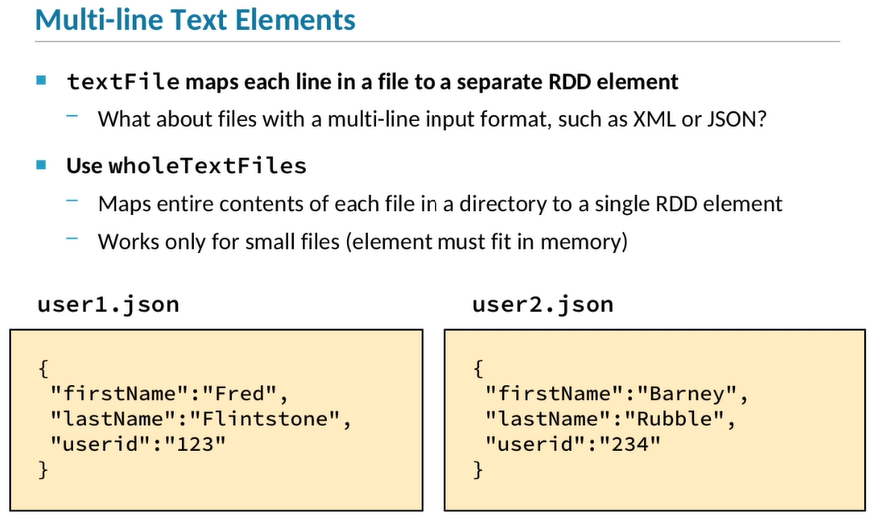


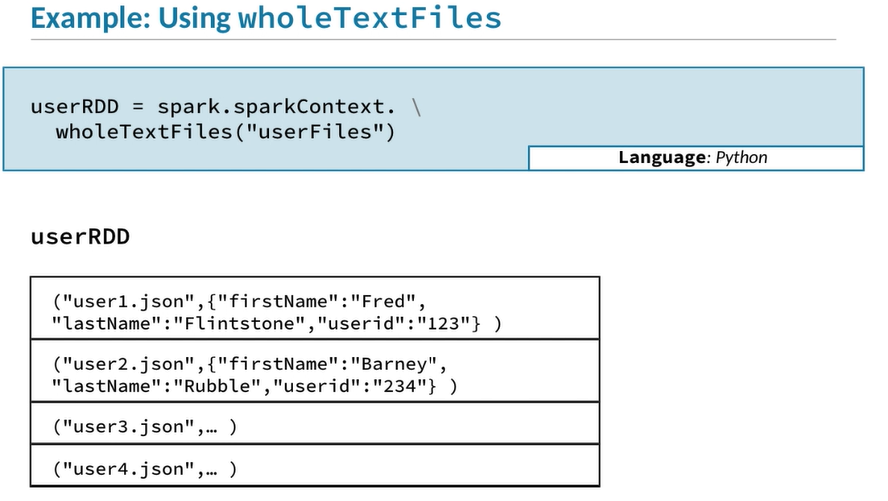
1. **The elements of an RDD can be objects of any type, as long as the object is serializable.**
2. This includes simple types like strings, integers, characters, booleans, and so on,
3. as well as collection types like sequences, lists, dictionaries, arrays, tuples, and so on.
4. You can even nest collections, such as using an array that includes tuples or other arrays.
5. RDDs can get as complex as you’d like.
6. RDDs can hold mixed types, meaning that each element of the RDD could be a different type.
7. Although general-purpose RDDs can hold any type of data, there are specialized types of RDDs that hold specific types of data.
8. Those RDDs have some additional functionality specifically applicable to that type of data.
9. For instance, pair RDDs consist of key-value pairs.
10. Pair RDDs are particularly useful for aggregating data across multiple elements, using the key to match those elements.
11. Double RDDs are another specific type of RDD;
12. these are RDDs which only contain numeric data.
13. These include operations for many kinds of statistical calculations, such as mean, standard deviation, and variance.
14. So where does the data in an RDD come from?
15. Like DataFrames and Datasets, each RDD is based on a data source.
16. RDDs support a number of different types and formats of data sources.
17. One common type of data source is text files, which might contain free-form text, JSON, XML, CSV, or other formats.
18. You can also programmatically create in-memory data and use that as the basis of an RDD.
19. You can create RDDs from other RDDs as well.
20. This is a key feature, because that’s how RDD transformations work.
21. Finally, you can create RDDs from Datasets and DataFrames.
22. RDDs are part of core Spark rather than Spark SQL.
23. As such, you’ll need to use the Spark application context rather than a Spark session to work with RDDs.
24. Every Spark application, including the Spark shell, has a single Spark context.
25. You can access the Spark context in a Spark SQL application using the sparkContext attribute of the session.
26. The Spark shell automatically assigns the Spark context to a variable called sc, so you can use that as well.
27. The SparkContext class has a few functions to create new RDDs from files.
28. There are two methods to create RDDs from text files.
29. The textFile function reads new-line delimited text files, turning each line of the file into a single RDD element,
30. whereas wholeTextFiles reads a set of files, with the content of each individual file becoming a single element in the RDD.
31. For files with other formats, you can use hadoopFile or newAPIHadoopFile.
32. These do the same thing, but use different versions of the Hadoop API.
33. The new API was introduced in Hadoop .20.
34. However, both are still in use, so Spark supports both for backward compatibility.

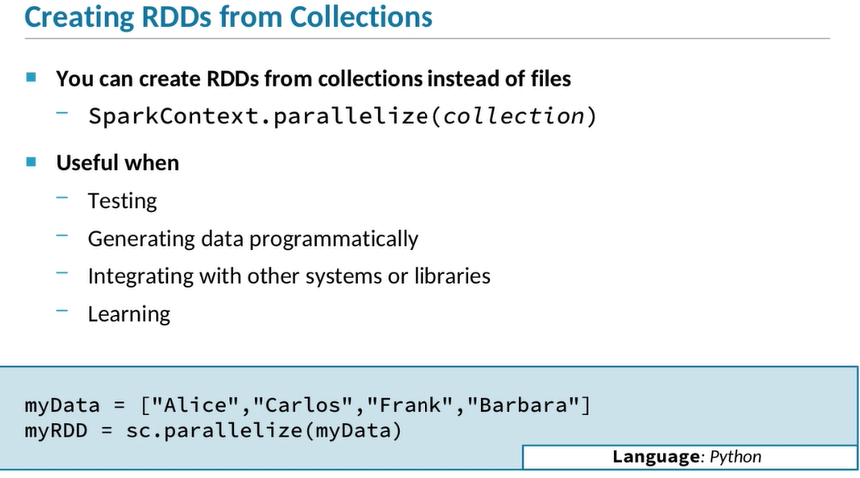
Creating and Saving RDDs

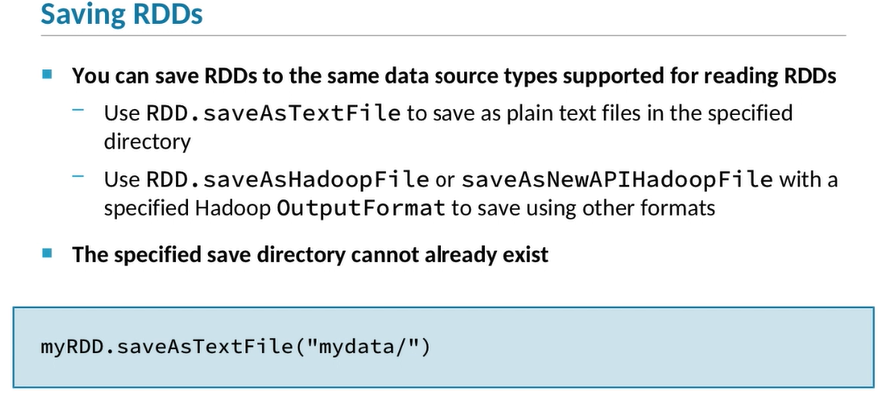






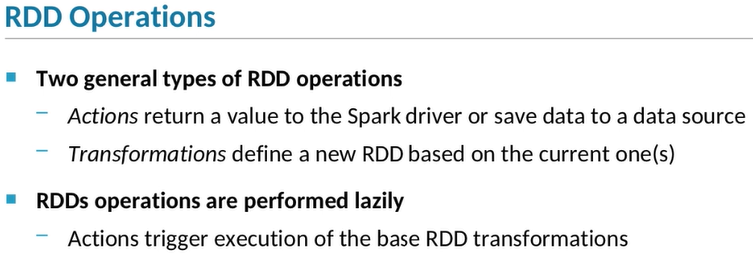


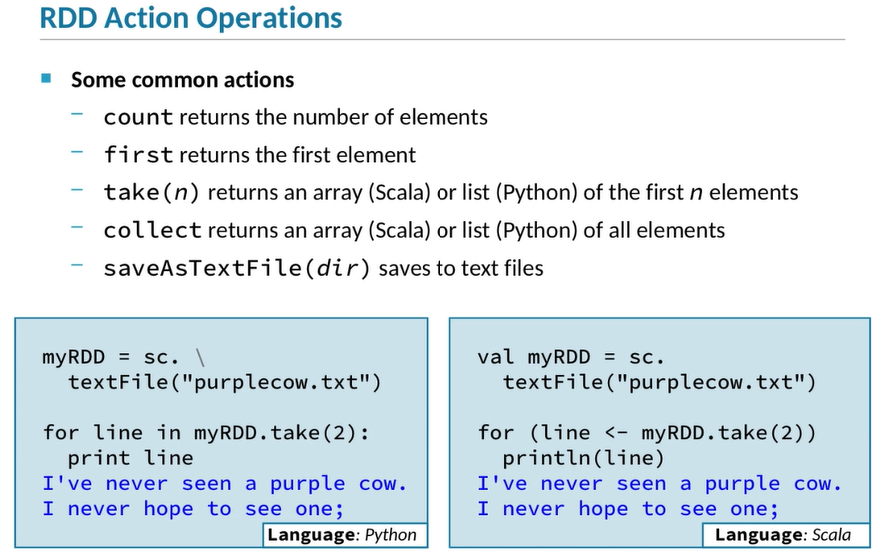




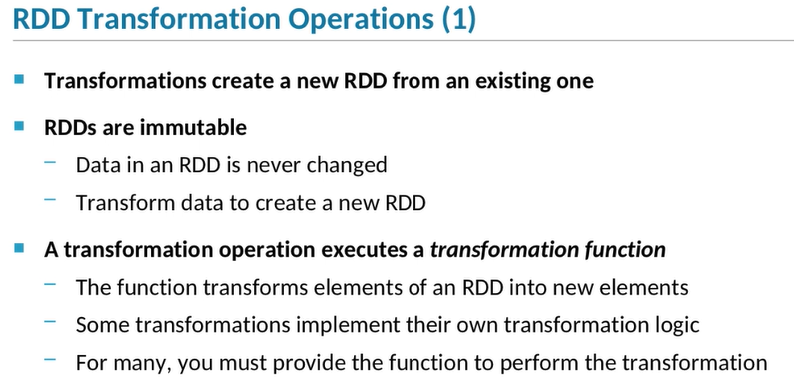
1. **Let’s look at how to create RDDs from text files, which is very common.**
2. One way to read text files is the sparkContext textFile method.
3. This method works specifically with text files where each line is terminated by a newline character.
4. You can use a single file, in which case, you can just pass the location and name of that file.
5. You can also use a set of files.
6. To load all the files in a directory, specify that directory path.
7. You can also use a wildcard or other regular expression to select only some of the files in a directory.
8. Or you can pass a string with a comma-separated list of individual files.
9. Just like with DataFrame and Dataset data sources, you can specify a full path name or a path relative to the configured file system’s default directory.
10. So in the example shown here, if the default file system is HDFS, Spark will look for the mydatadirectory under the current user's home directory.
11. The textFile method only supports text files in which every line is terminated by a newline character.
12. Each line becomes a single element in the RDD.
13. In the example here, we are starting with a data file called purplecow.txt, which contains four lines of a small poem.
14. When we read that data file into an RDD, we end up with four elements in the RDD,
15. each of which are strings containing the content of a single line of the file.
16. Although treating individual lines of a file as separate elements in the RDD is very common,
17. sometimes you’ll have files, such as XML or JSON files, in which a record spans multiple lines.
18. For instance, look at these examples from a set of JSON files.
19. Each file in the set contains a single user record containing firstName, lastName, and userid.
20. Each record spans multiple lines in the file.
21. The sparkContext wholeTextFiles function provides an easy way to work with single-record files like this.
22. It reads a set of files, converting the entire contents of each file to a single element in the RDD.
23. Note that this means that the entire file must be able to fit in memory.
24. So if you’ve got very large files, this may not be a good solution.
25. But if the files fit in memory, then this is a convenient way of reading whole files into individual elements of an RDD.
26. Here’s a code snippet demonstrating how to read multi-line elements from a set of text files.
27. We call wholeTextFiles and specify a directory or set of files, which reads each file in the set into a single element in the RDD.
28. This creates what we call a *pair RDD*.
29. A pair RDD has a key and a value for each element.
30. The wholeTextFiles method creates a pair RDD in which the key of the pair is the name of the file and the value of the pair is the entire contents of that file, as shown here.
31. Another way to create an RDD is from data in memory.
32. We can create an RDD from a collection or a list, for example, instead of by reading a file.
33. We do that using the parallelize method of the sparkContext.
34. This method is called parallelize because it turns the data into something than can be operated on by multiple tasks running in parallel.
35. Here’s an example in Python.
36. We create a variable myData, which is a list of these four strings: Alice, Carlos, Frank, and Barbara.
37. Then we say myRDD = sc.parallelize(myData), which creates an RDD from that list.
38. Creating an RDD from in-memory data is useful to create small RDDs for testing purposes.
39. It’s also useful if you need to generate your data programmatically in some way—for example, if you’re integrating with some other library to generate the data.
40. You can save RDDs to most of the same types of data sources that you can read from.
41. To save an RDD as text, call the RDD’s saveAsTextFile function.
42. This will create plain text files in which each line is the string representation of one element of the RDD.
43. Because Python, Java, and Scala all have the ability to convert any value to a string, you can save RDDs containing any type of value to text files.
44. To save an RDD using some other format call saveAsHadoopFile or saveAsNewAPIHadoopFilewith the OutputFormat class you want to use.
45. Note that, just as with DataFrames, you specify a directory path, not a specific file.
46. This is because RDDs and DataFrames are both typically distributed across multiple nodes.
47. Each chunk of data is called a partition, and each partition is saved into a separate file in the specified directory, using a name indicating which partition’s data is stored in the file.

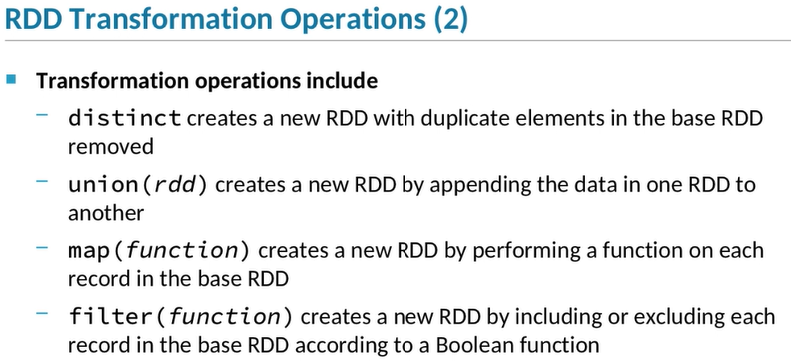
RDD Operations

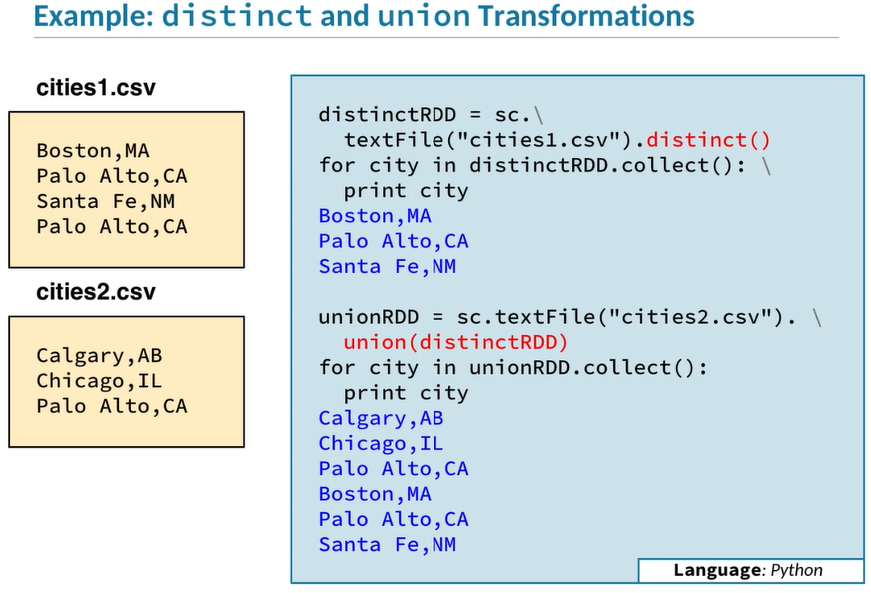




***Be Careful using Collect***







1. **There are two types of operations you can perform on RDDs: actions and transformations.**
2. An action generally returns a value from the application’s executors to the application driver or saves the data to a data source.
3. A transformation creates a new RDD based on the current RDD.
4. RDD transformations are executed lazily, similar to how DataFrame and Dataset queries are executed.
5. After a series of one or more transformations, an action is necessary to trigger the data to be loaded and the transformations to be executed.
6. Here are some of common RDD actions.
7. The count action returns the number of elements in an RDD.
8. The first action returns the first element.
9. The take action is similar to the first, but instead of returning a single element, it returns an array or list of the first specified number of elements.
10. So take(5) returns an array or list of the first five elements of the RDD.
11. Both first and take are useful when you’re debugging and you want to make sure the RDD contains the type of data that it should.
12. The collect action returns an array or list containing all of the elements in the RDD.
13. In most real-world applications, you should avoid calling collect, because it could potentially return a huge amount of data to the driver.
14. However, collect can be useful for testing or integration, or when you know the returned data is going to be fairly small.
15. Calling saveAsTextFile, of course, saves that RDD out to disk.
16. The Python and Scala examples on this slide both demonstrate the take action.
17. First, we create a new RDD based on the file purplecow.txt.
18. Then call take(2) to return an array of the first two elements of the RDD, which will be the first two lines in the file.
19. We loop through the resulting array, printing out each string element in the array, which correspond to the first two lines of the file.
20. The looping syntax is a bit different between Scala and Python, but both code snippets do the same thing: loop through and print out the first two lines of the poem.
21. The other type of operation you can perform on RDDs is a transformation.
22. The defining characteristic of a transformation is that it creates a new RDD based on an existing one—or in some cases, such as joining or appending, two existing ones.
23. This is an inherent part of the Spark architecture, because RDDs themselves are immutable; in other words, you can’t change the contents of an RDD once it exists.
24. Typically you create multiple RDDs, each one based on the previous one, until you get the final results that you’re looking for.
25. Transformation operations work by executing some function on the elements of an RDD to create the elements in the new RDD.
26. In some cases, the logic of the transformation function is embedded as part of the transformation operation itself.
27. In others, you must pass a function that implements the transformation logic you want to perform.
28. Here are some commonly used transformation operations.
29. The first one, distinct, creates a new RDD by removing all the duplicate elements in the base RDD.
30. The result is a new RDD in which each element is unique.
31. The union operation takes a second RDD, and appends the data in the second RDD with the first to create a new RDD.
32. The map and filter transformations are examples of operations that take a function as a parameter.
33. The map transformation applies the provided function to each element in the base RDD, which returns a new element to be included in the new RDD.
34. The filter transformation is similar,
35. but the function evaluates each element and returns true for elements that should be included in the new RDD and false for elements that shouldn't.
36. This list shows only a few of the many available RDD transformations.
37. Review the API documentation for the RDD class to see the full list.
38. Here we see an example of two different transformations: distinct and union.
39. We start with two datafiles, cities1.csv and cities2.csv, each containing a list of cities.
40. We start by reading the first datafile, then calling distinct to create a new RDD containing only the unique elements of the base RDD created by reading the file.
41. We call collect to return the data from the new RDD, and loop through the returned list, printing out each element.
42. Here we can see that although Palo Alto, California, occurs twice in the input datafile, it only occurs once in the RDD created by the distinct operation.
43. Next we read in the second datafile, cities2.csv, and call the union operation, passing distinctRDD, created above.
44. We again collect the data from the new RDD and display it, confirming that the new RDD contains all the elements of both input RDDs.