

Learning Spark Programming Basics

Talk is cheap. Show me the code.

Linus Torvalds, Finnish-American creator of Linux

In This Chapter:

- Resilient Distributed Datasets (RDDs)
- How to load data into Spark RDDs
- Transformation and actions on RDDs
- How to perform operations on multiple RDDs

Now that we've covered Spark's runtime architecture and how to deploy Spark, it's time to learn Spark programming in Python, starting with the basics. This chapter discusses a foundational concept in Spark programming and execution: Resilient Distributed Datasets (RDDs). You will also learn how to work with the Spark API, including the fundamental Spark transformations and actions and their usage. This is an intense chapter, but by the end of it, you will have all the basic building blocks you need to create any Spark application.

Introduction to RDDs

A Resilient Distributed Dataset (RDD) is the most fundamental data object used in Spark programming. RDDs are datasets within a Spark application, including

the initial dataset(s) loaded, any intermediate dataset(s), and the final resultant dataset(s). Most Spark applications load an RDD with external data and then create new RDDs by performing operations on the existing RDDs; these operations are *transformations*. This process is repeated until an output operation is ultimately required—for instance, to write the results of an application to a filesystem; these types of operations are *actions*.

RDDs are essentially distributed collections of objects that represent the data used in Spark programs. In the case of PySpark, RDDs consist of distributed Python objects, such as lists, tuples, and dictionaries. Objects within RDDs, such as elements in a list, can be of any object type, including primitive data types such as integers, floating point numbers, and strings, as well as complex types such as tuples, dictionaries, or other lists. If you are using the Scala or Java APIs, RDDs consist of collections of Scala and Java objects, respectively.

Although there are options for persisting RDDs to disk, RDDs are predominantly stored in memory, or at least they are intended to be stored in memory. Because one of the initial uses for Spark was to support machine learning, Spark's RDDs provided a restricted form of shared memory that could make efficient reuse of data for successive and iterative operations.

Moreover, one of the main downsides of Hadoop's implementation of MapReduce was its persistence of intermediate data to disk and the copying of this data between nodes at runtime. Although the MapReduce distributed processing method of sharing data did provide resiliency and fault tolerance, it was at the cost of latency. This design limitation was one of the major catalysts for the Spark project. As data volumes increased along with the necessity for real-time data processing and insights, Spark's mainly in-memory processing framework based on RDDs grew in popularity.

The term *Resilient Distributed Dataset* is an accurate and succinct descriptor for the concept. Here's how it breaks down:

- **Resilient:** RDDs are resilient, meaning that if a node performing an operation in Spark is lost, the dataset can be reconstructed. This is because Spark knows the lineage of each RDD, which is the sequence of steps to create the RDD.
- **Distributed:** RDDs are distributed, meaning the data in RDDs is divided into one or many partitions and distributed as in-memory collections of objects across Worker nodes in the cluster. As mentioned earlier in this chapter, RDDs provide an effective form of shared memory to exchange

data between processes (Executors) on different nodes (Workers).

- **Dataset:** RDDs are datasets that consist of records. Records are uniquely identifiable data collections within a dataset. A record can be a collection of fields similar to a row in a table in a relational database, a line of text in a file, or multiple other formats. RDDs are created such that each partition contains a unique set of records and can be operated on independently. This is an example of the *shared nothing* approach discussed in [Chapter 1](#).

Another key property of RDDs is their immutability, which means that after they are instantiated and populated with data, they cannot be updated. Instead, new RDDs are created by performing transformations such as `map` or `filter` functions, discussed later in this chapter, on existing RDDs.

Actions are the other operations performed on RDDs. Actions produce output that can be in the form of data from an RDD returned to a Driver program, or they can save the contents of an RDD to a filesystem (local, HDFS, S3, or other). There are many other actions as well, including returning a count of the number of records in an RDD.

[Listing 4.1](#) shows a sample Spark program loading data into an RDD, creating a new RDD using a `filter` transformation, and then using an action to save the resultant RDD to disk. We will look at each of these operations in this chapter.

Listing 4.1 Sample PySpark Program to Search for Errors in Log Files

[Click here to view code image](#)

```
# load log files from local filesystem
logfilesrdd = sc.textFile("file:///var/log/hadoop/hdfs/hadoop-hdfs-*.")
# filter log records for errors only
onlyerrorsrdd = logfilesrdd.filter(lambda line: "ERROR" in line)
# save onlyerrorsrdd as a file
onlyerrorsrdd.saveAsTextFile("file:///tmp/onlyerrorsrdd")
```

You can find more detail about RDD concepts in the University of California, Berkeley, paper “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing,”

<https://amplab.cs.berkeley.edu/publication/resilient-distributed-datasets-a-fault-tolerant-abstraction-for-in-memory-cluster-computing/>.

Loading Data into RDDs

RDDs are effectively created after they are populated with data. This can be the result of transformations on an existing RDD being written into a new RDD as part of a Spark program.

To start any Spark routine, you need to initialize at least one RDD with data from an external source. This initial RDD is then used to create further intermediate RDDs or the final RDD through a series of transformations and actions. The initial RDD can be created in several ways, including the following:

- Loading data from a file or files
- Loading data from a data source, such as a SQL or NoSQL datastore
- Loading data programmatically
- Loading data from a stream, as discussed in [Chapter 7, “Stream Processing and Messaging Using Spark”](#)

Creating an RDD from a File or Files

Spark provides API methods to create RDDs from a file, files, or the contents of a directory. Files can be of various formats, from unstructured text files, to semi-structured files such as JSON files, to structured data sources such as CSV files. Spark also supports several common serialized binary encoded formats, such as SequenceFiles and protocol buffers (protobufs), as well as columnar file formats such as Parquet and ORC (which we will discuss later).

Spark and File Compression

Spark includes native support for several lossless compression formats. Spark can seamlessly read from common compressed file formats, including GZIP and ZIP (or any other compressed archives created using the DEFLATE compression method), as well as BZIP2 compressed archives.

Spark also provides native codecs, which are libraries for compressing and decompressing data, that enable both reading and writing of compressed files. Built-in codecs include LZ4 and LZF, which are LZ77-based lossless compression formats, and Snappy.

Snappy is a fast, splittable, low-CPU data compression and decompression library from Google that is commonly used in the Hadoop core and ecosystem projects. Snappy is used by default for compressing data internal to Spark, such as the data in RDD partitions exchanged across the network between Workers.

Splittable Versus Non-splittable Compression Formats

It's important to distinguish between splittable and non-splittable compression formats when using distributed processing platforms such as Spark or Hadoop.

Splittable compression formats are indexed so they can split—typically on block boundaries—without compromising the integrity of the archive. Non-splittable formats are not indexed and cannot split. This means that a non-splittable archive must be readable in its entirety on one system because it cannot be distributed.

Although common desktop compression formats such as ZIP and GZIP can achieve high rates of compression, they are not splittable. This may be okay for small files containing lookup data, but for larger datasets, splittable compression formats such as Snappy or LZO are preferable. In some cases, you are better off decompressing files altogether before ingesting them into a distributed filesystem such as HDFS.

Data Locality with RDDs

By default, Spark tries to read data into an RDD from the nodes close to it. Because Spark usually accesses distributed partitioned data, such as data from HDFS or S3, to optimize transformation operations, it creates partitions to hold the underlying blocks from the distributed filesystem. [Figure 4.1](#) depicts how blocks from a file in a distributed filesystem such as HDFS are used to create RDD partitions on Workers, which are collocated with the data.

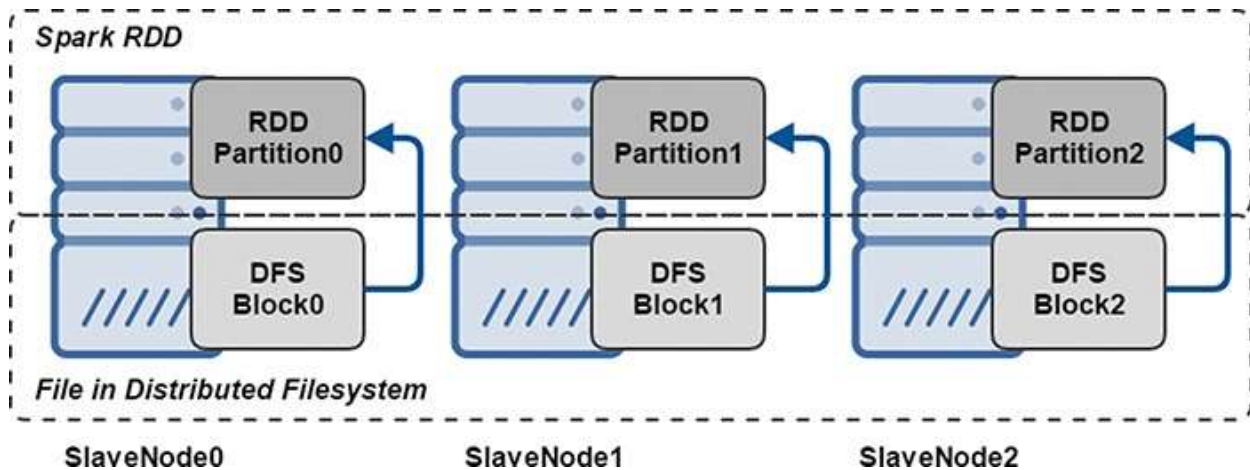


Figure 4.1 Loading an RDD from a text file in a distributed filesystem.

Loading RDDs from a Local Filesystem

If you are not using a distributed filesystem—for instance, if you are creating an RDD from a file on your local filesystem—you need to ensure that the file you are loading is available in the same relative path on all worker nodes in the cluster. Otherwise, you will get the following error:

[Click here to view code image](#)

```
java.io.FileNotFoundException: File does not exist
```

For this reason, it's preferable to use a distributed filesystem such as HDFS or S3 as a file-based source for Spark RDDs; in this case, you upload a file from your local filesystem to the distributed system first and then create the RDD from the distributed object, if possible. Another alternative approach to using a local filesystem is to use a shared network filesystem instead.

Methods for Creating RDDs from a Text File or Files

The Spark methods for creating an RDD from a file or files support several filesystems. The scheme in the URI specifies these filesystems. This scheme is the prefix followed by `://`. You see this all the time with Internet resources referred to by the scheme `http://` or `https://`. [Table 4.1](#) summarizes schemes and URI structures for common filesystems supported by Spark.

Table 4.1 Filesystem Schemes and URI Structures

Filesystem	URI Structure
------------	---------------

Local filesystem	file:///path
HDFS*	hdfs://hdfs_path
Amazon S3*	s3://bucket/path (also used are s3a and s3n)
OpenStack Swift*	swift://container.PROVIDER/path

* Requires filesystem configuration parameters to be set.

You can use text files and the methods described in the following sections to create RDDs.

textFile()

Syntax:

[Click here to view code image](#)

```
sc.textFile(name, minPartitions=None, use_unicode=True)
```

The `textFile()` method is used to create RDDs from files (compressed or uncompressed), directories, or glob patterns (file patterns with wildcards).

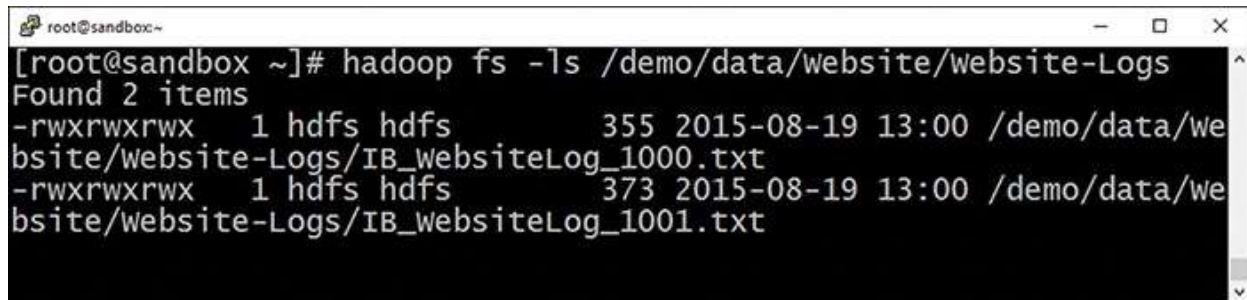
The `name` argument specifies the path or glob to be referenced, including the filesystem scheme, as shown in [Table 4.1](#).

The `minPartitions` argument determines the number of partitions to create. By default, if you are using HDFS, each block of the file (typically 128MB) creates a single partition, as demonstrated in [Figure 4.1](#). You can request more partitions than there are blocks; however, any number of partitions specified that is less than the number of blocks will revert to the default behavior of one block to one partition.

The `use_unicode` argument specifies whether to use Unicode or UTF-8 as the character encoding scheme.

The `minPartitions` and `use_unicode` arguments are optional as they have default values configured. In most cases, it's not necessary to supply these parameters explicitly unless you need to override the defaults.

Consider the Hadoop filesystem directory shown in [Figure 4.2](#).

A terminal window titled 'root@sandbox:~' showing the command 'hadoop fs -ls /demo/data/website/website-Logs' and its output. The output lists two files: 'IB_websiteLog_1000.txt' (355 bytes) and 'IB_websiteLog_1001.txt' (373 bytes), both dated 2015-08-19 13:00. The permissions are '-rwxrwxrwx' and the owner is 'hdfs' for both.

```
[root@sandbox ~]# hadoop fs -ls /demo/data/website/website-Logs
Found 2 items
-rwxrwxrwx  1 hdfs hdfs          355 2015-08-19 13:00 /demo/data/we
bsite/website-Logs/IB_websiteLog_1000.txt
-rwxrwxrwx  1 hdfs hdfs          373 2015-08-19 13:00 /demo/data/we
bsite/website-Logs/IB_websiteLog_1001.txt
```

Figure 4.2 HDFS directory listing.

To read files in HDFS from Spark, the `HADOOP_CONF_DIR` environment variable must be set on all worker nodes of the cluster. The Hadoop config directory contains all the information used by Spark to connect to the appropriate HDFS cluster. You can set this automatically by using the `spark-env.sh` script located in the `conf/` directory of each Spark installation. The command used to set this variable on Linux systems is as follows:

[Click here to view code image](#)

```
export HADOOP_CONF_DIR=/etc/hadoop/conf
```

[Listing 4.2](#) provides examples of the `textFile()` method loading the data from the HDFS directory pictured in [Figure 4.2](#).

Listing 4.2 Creating RDDs Using the `textFile()` Method

[Click here to view code image](#)

```
# load the contents of the entire directory
logs = sc.textFile("hdfs:///demo/data/Website/Website-Logs/")
# load an individual file
logs = sc.textFile("hdfs:///demo/data/Website/Website-
Logs/IB_websiteLog_1001.txt")
# load a file or files using a glob pattern
logs = sc.textFile("hdfs:///demo/data/Website/Website-Logs/*_1001.txt")
```

In each of the examples in [Listing 4.2](#), an RDD named `logs` is created, with each line of the file represented as a record.

`wholeTextFiles()`

Syntax:

[Click here to view code image](#)

```
sc.wholeTextFiles(path, minPartitions=None, use_unicode=True)
```

The `wholeTextFiles()` method lets you read a directory containing multiple files. Each file is represented as a record consisting of a key containing the filename and a value containing the contents of the file. In contrast, when reading all files in a directory with the `textFile()` method, each line of each file represents a separate record with no context of the line's file origin.

Typically with event processing, the originating filename is not required because the record contains a timestamp field.

As each record contains the contents of the entire file with the `wholeTextFiles()` method, this method is intended for use with small files. The `minPartitions` and `use_unicode` arguments behave similarly to the `textFile()` method.

Using the HDFS directory shown in [Figure 4.2](#), [Listing 4.3](#) provides an example of the `wholeTextFiles()` method.

Listing 4.3 Creating RDDs by Using the `wholeTextFiles()` Method

[Click here to view code image](#)

```
# load the contents of the entire directory into key/value pairs
logs = sc.wholeTextFiles("hdfs:///demo/data/Website/Website-Logs/")
```

To demonstrate the difference between the `textFile()` and `wholeTextFiles()` methods in Spark, let's look at an example. This is an example you can try for yourself on any Spark installation. The Spark installation includes a directory named `licenses` that contains license files for all the open source projects used within the Spark project (for example, Scala).

Using the `licenses` directory as a source of text files to load different RDDs, [Listing 4.4](#) shows the difference between the `textFile()` and `wholeTextFiles()` methods.

Listing 4.4 Comparing the `textFile()` and `wholeTextFiles()` Methods

[Click here to view code image](#)

```
# load the contents of the entire directory into an RDD named
licensefiles
licensefiles = sc.textFile("file:///opt/spark/licenses/")
# inspect the object created
licensefiles
# returns:
#   file:///opt/spark/licenses/ MapPartitionsRDD[1] at textFile at
#   NativeMethodAccessorImpl.java:0
licensefiles.take(1)
# returns a list containing the first line of the first file read in the
directory:
# [u'The MIT License (MIT)']
licensefiles.getNumPartitions()
# there is a partition created for each file in the directory, in this
case the
# return value is 36
licensefiles.count()
# this action will count the combined total number of lines in all the
files, in
# this case the return value is 1075

# now let's perform a similar exercise using the same directory using
the
# wholeTextFiles() method instead
licensefile_pairs = sc.wholeTextFiles("file:///opt/spark/licenses/") #
inspect the object created
licensefile_pairs
# returns:
#   org.apache.spark.api.java.JavaPairRDD@3f714d2d
licensefile_pairs.take(1)
# returns the first key/value pair as a list of tuples, with the key
being each file # and the value being the entire contents of that file:
# [(u'file:/opt/spark/licenses/LICENSE-scopt.txt', u'The MIT License
(MIT)\n...)..]
licensefile_pairs.getNumPartitions()
# this method will create a single partition (1) containing key/value
pairs for each # file in the directory
licensefile_pairs.count()
# this action will count the number of files or key/value pairs
```

in this case the return value is 36

Creating an RDD from an Object File

Spark supports several common object file implementations. The term *object files* refers to serialized data structures that are not normally human readable and that are designed to provide structure and context to data, making access to data more efficient for the requesting platform.

Sequence files are encoded serialized files commonly used in Hadoop. You can create RDDs by using the `sequenceFile()` method. There is also a similar method called `hadoopFile()`. (For brevity, we won't cover sequence files in detail in this book because it would require more knowledge about serialization in Hadoop, which is beyond the book's scope.)

In addition, there is support for reading and writing Pickle files, a special serialization format for Python. Similar functionality is available for serialized Java objects with the `objectFile()` method.

Spark also has native support for JSON files, which we will look at shortly.

Creating an RDD from a Data Source

It is commonly required to load data from a database into an RDD in a Spark program as a source of historical data, master data, or reference or lookup data. This data can come from a variety of host systems and database platforms, including Oracle, MySQL, Postgres, and SQL Server.

As with the creation of RDDs using external files, RDDs created using data from an external database—a MySQL database, for example—attempt to move the data into multiple partitions across multiple Workers. This maximizes parallelism during processing, especially during the initial stages. In addition, if you divide the table, typically by key space, into different partitions, the partitions can load in parallel as well, and each partition is responsible for fetching a unique set of rows. This concept is depicted in [Figure 4.3](#).

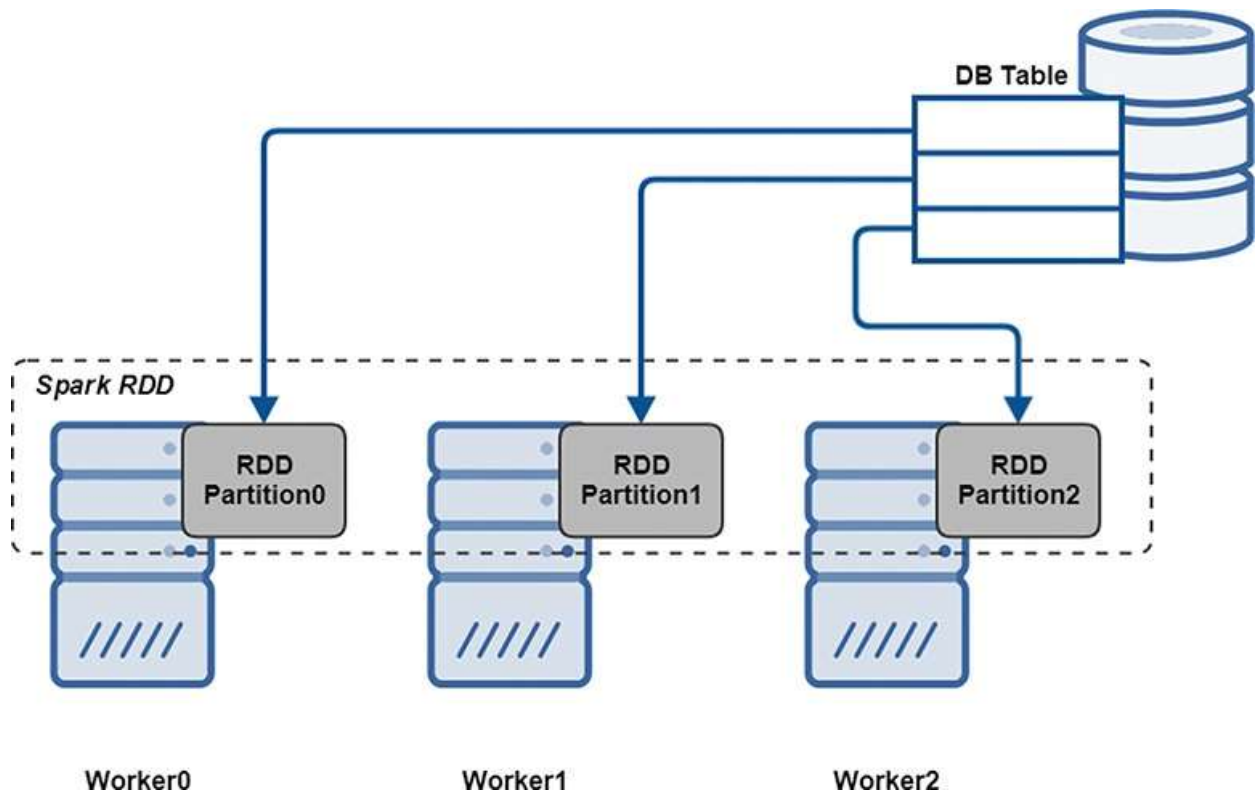


Figure 4.3 Loading an RDD from a table in a relational database.

The preferred methods of creating an RDD from a relational database table or query involve using functions from the `SparkSession` object. Recall that this is the main entry point for working with all types of data in Spark, including tabular data. The `SparkSession` exposes a `read` function, which returns a `DataFrameReader` object. You can then use this object to read data in a `DataFrame`, a special type of RDD previously referred to as a `SchemaRDD`. ([Chapter 6, “SQL and NoSQL Programming with Spark,”](#) covers `DataFrames` in more detail.)

The `read()` method has a `jdbc` function that can connect to and collect data from any JDBC-compliant data source.

Java Database Connectivity (JDBC)

Java Database Connectivity (JDBC) is a popular Java API for accessing different (mainly relational) database management systems (DBMSs), managing functions such as connecting to and disconnecting from a DBMS and running queries. Database vendors typically provide drivers or connectors to provide access to their database platforms via JDBC.

Because Spark processes run in Java virtual machines (JVMs), JDBC is natively available to Spark.

Consider a MySQL Server called `mysqlserver` with a database named `employees` with a table called `employees`. The `employees` table has a primary key named `emp_no` that is a logical candidate to use for dividing the key space from the table into multiple partitions. To access the MySQL database via JDBC, you need to launch `pyspark` providing the `mysql-connector.jar` in the driver class path. Connectors, such as the `mysql-connector.jar`, are generally available from your target database platform vendor's website. An example of this is shown in [Listing 4.5](#).

Listing 4.5 Launching `pyspark` and Supplying the JDBC MySQL Connector JAR File

[Click here to view code image](#)

```
# download the latest jdbc connector for your target database, include
as follows:
$SPARK_HOME/bin/pyspark \
--driver-class-path mysql-connector-java-5.*-bin.jar \
--master local
```

Once you have launched an interactive or non-interactive Spark application, including the relevant JDBC connection library for your target database, you can use the `jdbc` method of the `DataFrame` reader object.

`read.jdbc()`

Syntax:

[Click here to view code image](#)

```
spark.read.jdbc(url, table,
                 column=None,
                 lowerBound=None,
                 upperBound=None,
                 numPartitions=None,
                 predicates=None,
                 properties=None)
```

The `url` and `table` arguments specify the target database and table to read.

The `column` argument helps Spark choose the appropriate column, preferably a `long` or `int` datatype, to create the number of partitions specified by `numPartitions`. The `upperBound` and `lowerBound` arguments are used in conjunction with the `column` argument to assist Spark in creating the partitions. These represent the minimum and maximum values for the specified column in the source table. If any one of these arguments is supplied with the `read.jdbc()` function, all must be supplied.

The optional argument `predicates` allows for including `WHERE` conditions to filter unneeded records while loading partitions. You can use the `properties` argument to pass parameters to the JDBC API, such as the database user credentials; if supplied, this argument must be a Python dictionary, a set of name/value pairs representing the various configuration options.

[Listing 4.6](#) shows the creation of an RDD using the `read.jdbc()` method.

Listing 4.6 Loading Data from a JDBC Data Source by Using `read.jdbc()`

[Click here to view code image](#)

```
employeesdf =
spark.read.jdbc(url="jdbc:mysql://localhost:3306/employees",

table="employees",column="emp_no",numPartitions="2",lowerBound="10001",
    upperBound="499999",properties={"user":"<user>","password":"<pwd>"})
employeesdf.rdd.getNumPartitions()
# should return 2 as we specified numPartitions=2
```

The `read.jdbc()` function returns a `DataFrame` (a special Spark object against which SQL queries can be executed), as shown in [Listing 4.7](#).

Listing 4.7 Running SQL Queries Against Spark DataFrames

[Click here to view code image](#)

```
sqlContext.registerDataFrameAsTable(employeesdf, "employees")
df2 = spark.sql("SELECT emp_no, first_name, last_name FROM employees
LIMIT 2")
```

```
df2.show()
# will return a 'pretty printed' result set similar to:
#+-----+-----+-----+
#|emp_no|first_name|last_name|
#+-----+-----+-----+
#| 10001|    Georgi|  Facello|
#| 10002|   Bezalel|   Simmel|
#+-----+-----+-----+
```

Creating Too Many Partitions Using the `read.jdbc()` Function

Be careful not to specify too many partitions when loading a DataFrame from a relational data source. Each partition running on each individual worker node connects to the DBMS independently and queries its designated portion of the dataset. If you have hundreds or thousands of partitions, this could be misconstrued as a distributed denial-of-service (DDoS) attack on the host database system.

Creating RDDs from JSON Files

JSON (JavaScript Object Notation) is a popular data-interchange format. JSON is a “self-describing” format, which is human readable and commonly used to return responses from web services and RESTful APIs. JSON objects are treated as data sources and accessed using the `read.json()` method that is exposed through the SparkSession entry point.

`read.json()`

Syntax:

[Click here to view code image](#)

```
spark.read.json(path, schema=None)
```

The `path` argument specifies the full path to the JSON file you are using as a data source. You can use the optional `schema` argument to specify the target schema for creating the DataFrame.

Consider a JSON file named `people.json` that contains the names and,

optionally, the ages of people. This file happens to be in the `examples` directory of the Spark installation, as shown in [Figure 4.4](#).



Figure 4.4 JSON file.

[Listing 4.8](#) demonstrates the creation of a DataFrame named `people` using the `people.json` file.

Listing 4.8 Creating and Working with a DataFrame Created from a JSON File

[Click here to view code image](#)

```
people =
spark.read.json("/opt/spark/examples/src/main/resources/people.json")
# inspect the object created
people
# notice that a DataFrame is created which includes the following
schema:
# DataFrame[age: bigint, name: string]
# this schema was inferred from the object
people.dtypes
# the dtypes method returns the column names and datatypes in this case
it returns:
#[('age', 'bigint'), ('name', 'string')]
people.show()
# you should see the following output
```



```

#+-----+-----+
#| age|    name|
#+-----+-----+
#|null|Michael|
#| 30|    Andy|
#| 19|   Justin|
#+-----+-----+
# as with all DataFrames you can create use them to run SQL queries as
follows
sqlContext.registerDataFrameAsTable(people, "people")
df2 = spark.sql("SELECT name, age FROM people WHERE age > 20")
df2.show()
# you should see the resultant output below
#+-----+-----+
#|name|age|
#+-----+-----+
#|Andy| 30|
#+-----+-----+

```

Creating an RDD Programmatically

It is possible to create an RDD programmatically from data in your program, whether the data is in lists, arrays, or collections. The data from your collection is partitioned and distributed in much the same way as it is using the previous methods. However, creating RDDs this way can be limiting because it requires all of the dataset to exist or be created in memory on one system. The following sections describe methods exposed by the SparkContext that allow you to create RDDs from lists in your program.

parallelize()

Syntax:

[Click here to view code image](#)

```
sc.parallelize(c, numSlices=None)
```

The `parallelize()` method assumes that you have a list created already and that you supply it as the `c` (for collection) argument. The `numSlices` argument specifies the desired number of partitions to create. An example of the

`parallelize()` method is shown in [Listing 4.9](#).

Listing 4.9 Creating an RDD by Using the `parallelize()` Method

[Click here to view code image](#)

```
parallelrdd = sc.parallelize([0, 1, 2, 3, 4, 5, 6, 7, 8])
parallelrdd
# notice the type of RDD created:
# ParallelCollectionRDD[0] at parallelize at PythonRDD.scala:423
parallelrdd.count()
# this action will return 9 as this is the number of elements in our
collection
parallelrdd.collect()
# will return the parallel collection as a list as follows:
# [0, 1, 2, 3, 4, 5, 6, 7, 8]
```

`range()`

Syntax:

[Click here to view code image](#)

```
sc.range(start, end=None, step=1, numSlices=None)
```

The `range()` method generates a list for you and creates and distributes the RDD. The `start`, `end`, and `step` arguments define the sequence of values, and `numSlices` specifies the desired number of partitions. An example of the `range()` method is shown in [Listing 4.10](#).

Listing 4.10 Creating an RDD by Using the `range()` Method

[Click here to view code image](#)

```
# create an RDD with 1000 integers starting at 0 in increments of 1
# across 2 partitions
range_rdd = sc.range(0, 1000, 1, 2)
range_rdd
# note the PythonRDD type, as range is a native Python function
# PythonRDD[1] at RDD at PythonRDD.scala:43
range_rdd.getNumPartitions()
```

```
# should return 2 as we requested numSlices=2 range_rdd.min()
# should return 0 as this was out start argument
range_rdd.max()
# should return 999 as this is 1000 increments of 1 starting from 0
range_rdd.take(5)
# should return [0, 1, 2, 3, 4]
```

Operations on RDDs

Now that you have learned how to create RDDs from files in various filesystems, from relational data sources, and programmatically, let's look at the types of operations you can perform against RDDs and some of the key RDD concepts.

Key RDD Concepts

Recall that *transformations* in Spark are functions that operate on an RDD and return a new RDD, whereas *actions* operate against an RDD and return a value or perform an output operation. We will look at many examples of both shortly, but first we need to introduce two concepts: coarse-grained transformations and lazy evaluation.

Coarse-Grained Versus Fine-Grained Transformations

Operations performed against RDDs are considered to be coarse grained as they apply a function (a `map` or `filter` function, for example, which we will discuss shortly) against every element in the dataset, and they return a new dataset with the transformations applied. In contrast to coarse-grained transformations, fine-grained transformations can manipulate a single record or data cell, such as single-row updates in a relational database or `put` operations in a NoSQL database. Coarse-grained transformations are conceptually similar to Hadoop's implementation of the MapReduce programming model.

Transformations, Actions, and Lazy Evaluation

Transformations are operations performed against RDDs that result in the creation of new RDDs. Common transformations include `map` and `filter`

functions. The following example shows a new RDD created from a transformation of an existing RDD:

[Click here to view code image](#)

```
originalrdd = sc.parallelize([0, 1, 2, 3, 4, 5, 6, 7, 8])
newrdd = originalrdd.filter(lambda x: x % 2)
```

`originalrdd` originated from a parallelized collection of numbers. The `filter()` transformation was then applied to each element in the `originalrdd` to bypass even numbers in the collection. This transformation results in the RDD called `newrdd`.

In contrast to transformations, which return new RDD objects, actions return values or data to the driver program. Common actions include `reduce()`, `collect()`, `count()`, and `saveAsTextFile()`. The following example uses the `collect()` action to display the contents of `newrdd`:

[Click here to view code image](#)

```
newrdd.collect() # will return [1, 3, 5, 7]
```

Spark uses *lazy evaluation*, also called *lazy execution*, in processing Spark programs. Lazy evaluation defers processing until an action is called (that is, when output is required). This is easily demonstrated using an interactive shell, where you can enter one or more transformation methods to RDDs one after the other without any processing starting. Instead, each statement is parsed for syntax and object references only. After requesting an action such as `count()` or `saveAsTextFile()`, a DAG is created along with logical and physical execution plans. The Driver then orchestrates and manages these plans across Executors.

This lazy evaluation allows Spark to combine operations where possible, thereby reducing processing stages and minimizing the amount of data transferred between Spark Executors in a process called *shuffling*.

RDD Persistence and Reuse

RDDs are created and exist predominantly in memory on Executors. By default, RDDs are transient objects that exist only while they are required. After they transform into new RDDs and aren't needed for any other operations, they are removed permanently. This may be problematic if an RDD is required for more than one action because it must be reevaluated in its entirety each time. An option to address this is to cache or persist the RDD by using the `persist()`

method. Listings 4.11 and 4.12 demonstrate the effects of persisting an RDD.

Listing 4.11 Using an RDD for Multiple Actions Without Persistence

[Click here to view code image](#)

```
numbers = sc.range(0, 1000000, 1, 2)
evens = numbers.filter(lambda x: x % 2)
noelements = evens.count()
# processes evens RDD
print "There are %s elements in the collection" % (noelements)
# returns "There are 500000 elements in the collection"
listofelements = evens.collect()
# REPROCESSES evens RDD
print "The first five elements include " + (str(listofelements[0:5]))
# returns "The first five elements include [1, 3, 5, 7, 9]"
```

Listing 4.12 Using an RDD for Multiple Actions with Persistence

[Click here to view code image](#)

```
numbers = sc.range(0, 1000000, 1, 2)
evens = numbers.filter(lambda x: x % 2)
evens.persist()
# instructs Spark to persist evens RDD when the next action requires it
noelements = evens.count()
# processes and persists evens RDD in memory
print "There are %s elements in the collection" % (noelements)
# returns "There are 500000 elements in the collection"
listofelements = evens.collect()
# does NOT have to recompute the evens RDD
print "The first five elements include " + (str(listofelements[0:5]))
# returns "The first five elements include [1, 3, 5, 7, 9]"
```

After a request to persist the RDD using the `persist()` method (note that there is a similar `cache()` method as well), the RDD remains in memory on all the nodes in the cluster where it is computed after the first action called on it. You can see the persisted RDD in your Spark application UI in the Storage tab,

as shown in [Figure 4.5](#).

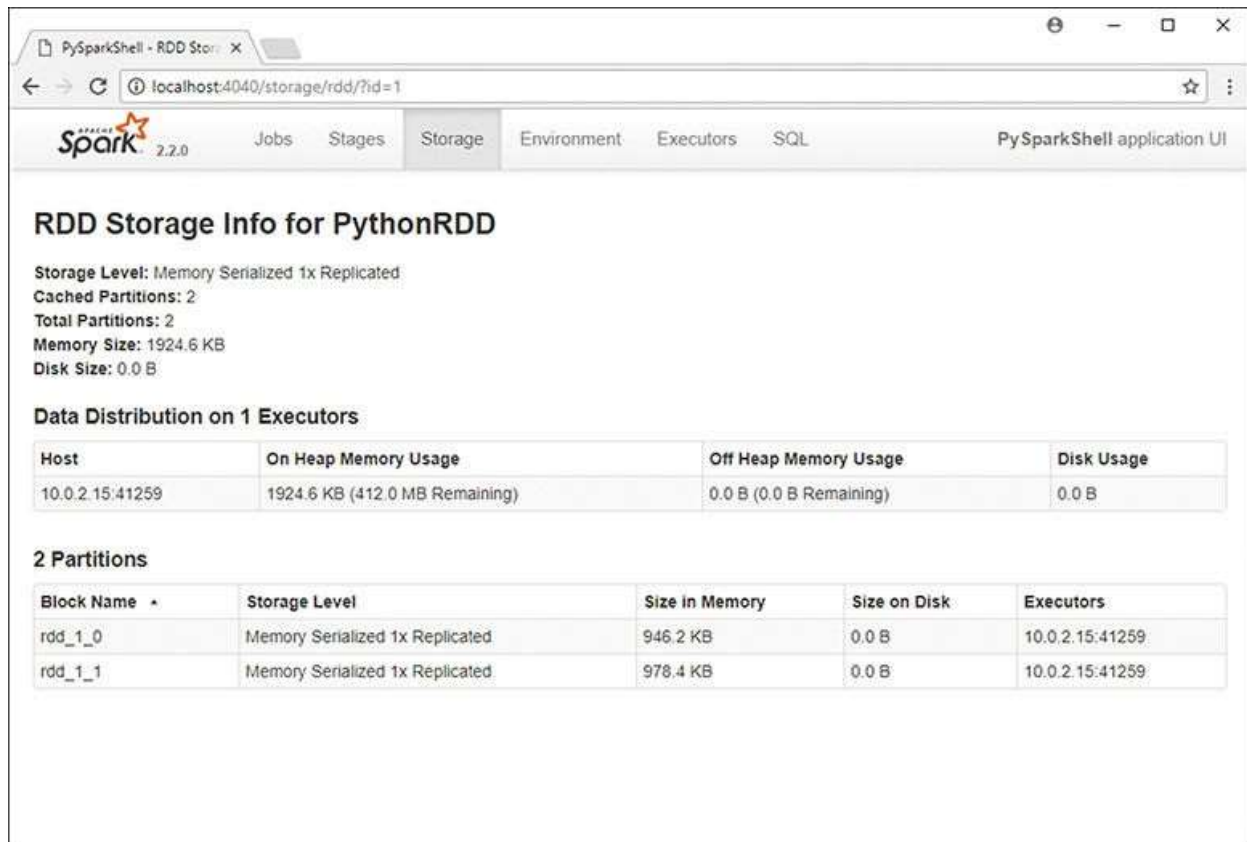


Figure 4.5 Storage tab in the Spark application UI.

RDD Lineage

Spark keeps track of each RDD's lineage—that is, the sequence of transformations that resulted in the RDD. As discussed previously, every RDD operation recomputes the entire lineage by default unless RDD persistence is requested.

In an RDD's lineage, each RDD has a parent RDD and/or a child RDD. Spark creates a directed acyclic graph (DAG) consisting of dependencies between RDDs. RDDs are processed in stages, which are sets of transformations. RDDs and stages have dependencies that can be narrow or wide.

Narrow dependencies, or *narrow operations*, are categorized by the following traits:

- Operations can collapse into a single stage; for instance, a `map()` and `filter()` operation against elements in the same dataset can be

processed in a single pass of each element in the dataset.

- Only one child RDD depends on the parent RDD; for instance, an RDD is created from a text file (the parent RDD), with one child RDD to perform the set of transformations in one stage.
- No shuffling of data between nodes is required.

Narrow operations are preferred because they maximize parallel execution and minimize shuffling, which is quite expensive and can be a bottleneck.

Wide dependencies of wide *operations*, in contrast, have the following traits:

- Operations define new stages and often require shuffling.
- RDDs have multiple dependencies; for instance, a `join()` operation (covered shortly) requires an RDD to be dependent upon two or more parent RDDs.

Wide operations are unavoidable when grouping, reducing, or joining datasets, but you should be aware of the impacts and overhead involved with these operations.

Lineage can be visualized by using the DAG Visualization option link from the Jobs or Stages detail page in the Spark application UI. [Figure 4.6](#) shows a DAG with multiple stages as a result of a wide operation (`reduceByKey()` in this case).

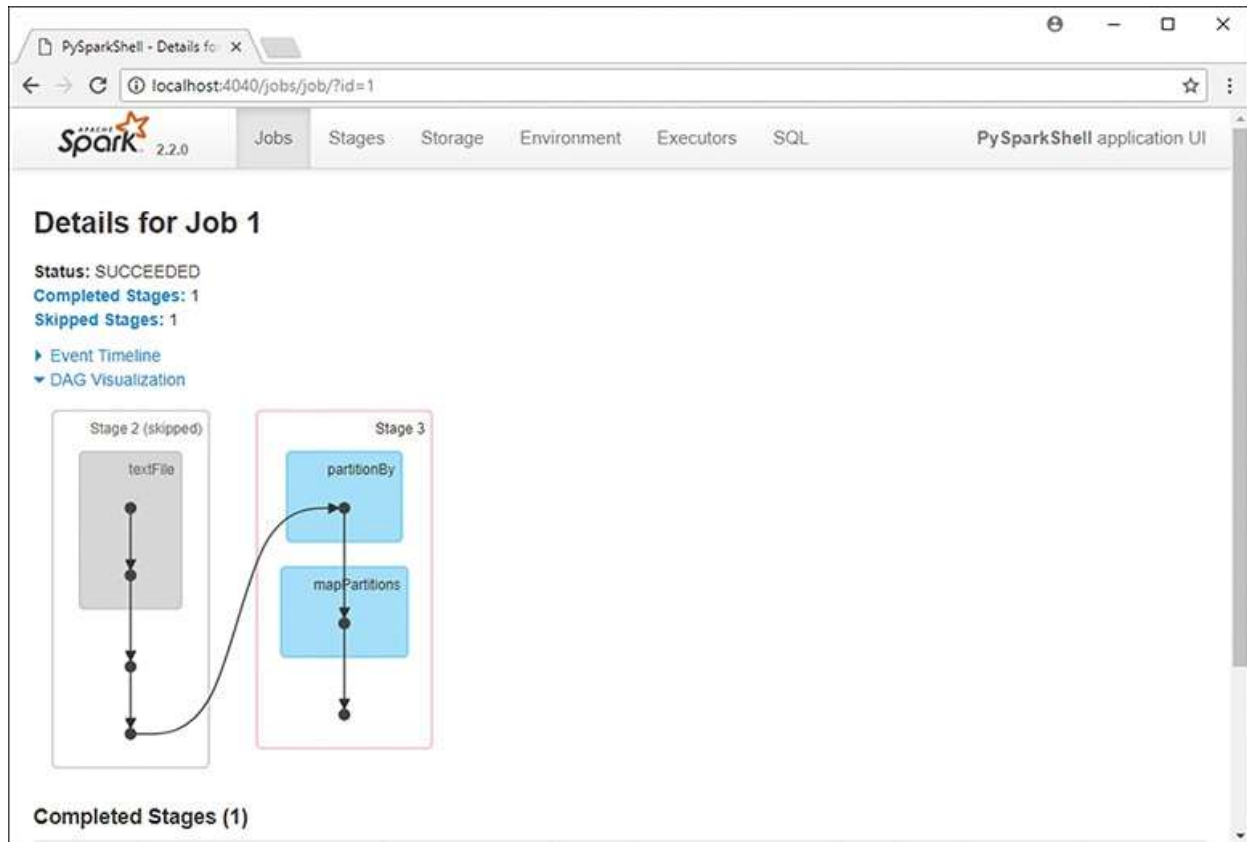


Figure 4.6 DAG visualization in the Spark application UI.

Fault Tolerance with RDDs

Spark records the lineage of each RDD, including the lineage of all parent RDDs and parents' parents, and so on. Any RDD with all of its partitions can be reconstructed to the state it was in at the time of the failure, which could have resulted from a node failure, for example. Because RDDs are distributed, they can tolerate and recover from the failure of any single node.

Non-deterministic Functions and Fault Tolerance

The use of non-deterministic functions in a Spark program—that is, functions that can produce different output given the same inputs, such as `random()`—will impact the ability to re-create RDDs in a consistent, repeatable state. This is further complicated if you use the non-deterministic function as a condition, which affects the logic or flow of the program. Use caution when implementing non-deterministic functions.

You can avert long recovery periods for complex processing operations by

checkpointing, or saving the data to a persistent file-based object. ([Chapter 5, “Advanced Programming Using the Spark Core API,”](#) discusses checkpointing.)

Types of RDDs

Aside from the base RDD class that contains members (properties or attributes and functions) common to all RDDs, there are some specific RDD implementations that enable additional operators and functions. These additional RDD types include the following:

- **PairRDD:** An RDD of key/value pairs. You have already seen this type of RDD as it is automatically created by using the `wholeTextFiles()` method.
- **DoubleRDD:** An RDD consisting of a collection of double values only. Because the values are of the same numeric type, several additional statistical functions are available, including `mean()`, `sum()`, `stdev()`, `variance()`, and `histogram()`, among others.
- **DataFrame (formerly known as SchemaRDD):** A distributed collection of data organized into named and typed columns. A DataFrame is equivalent to a relational table in Spark SQL. DataFrames originated with the `read.jdbc()` and `read.json()` functions discussed earlier.
- **SequenceFileRDD:** An RDD created from a SequenceFile, either compressed or uncompressed.
- **HadoopRDD:** An RDD that provides core functionality for reading data stored in HDFS using the v1 MapReduce API.
- **NewHadoopRDD:** An RDD that provides core functionality for reading data stored in Hadoop—for example, files in HDFS, sources in HBase, or S3—using the new MapReduce API (`org.apache.hadoop.mapreduce`).
- **CoGroupedRDD:** An RDD that cogroups its parents. For each key in parent RDDs, the resulting RDD contains a tuple with the list of values for that key. (We will discuss the `cogroup()` function later in this chapter.)
- **JdbcRDD:** An RDD resulting from a SQL query to a JDBC connection. It is available in the Scala API only.
- **PartitionPruningRDD:** An RDD used to prune RDD partitions or other partitions to avoid launching tasks on all partitions. For example, if you know the RDD is partitioned by range, and the execution DAG has a filter

on the key, you can avoid launching tasks on partitions that don't have the range covering the key.

- **ShuffledRDD:** The resulting RDD from a shuffle, such as repartitioning of data.
- **UnionRDD:** An RDD resulting from a `union()` operation against two or more RDDs.

There are other RDD variants, including `ParallelCollectionRDD` and `PythonRDD`, which are created from the `parallelize()` and `range()` functions discussed previously.

Throughout this book, in addition to the base RDD class, you will mainly use the `PairRDD`, `DoubleRDD`, and `DataFrame` RDD classes, but it's worthwhile to be familiar with all the various RDD types. Documentation and more information about the types of RDDs can be found in the Spark Scala API documentation at <https://spark.apache.org/docs/latest/api/scala/index.html>.

Basic RDD Transformations

The most commonly used Spark functions include `map()`, `flatMap()`, `filter()`, and `distinct()`, which are covered in the following sections. You will also learn about the `groupBy()` and `sortBy()` functions, which are commonly implemented by other functions. Grouping data is a normal precursor to performing aggregation or summary functions such as summing, counting, and so on. Sorting data is another useful operation for preparing output or for looking at the top or bottom records in a dataset. The `groupBy()` and `sortBy()` functions should be familiar to you if you have experience in relational database programming because they are analogous to the `GROUP BY` and `ORDER BY` functions in SQL.

`map()`

Syntax:

[Click here to view code image](#)

```
RDD.map(<function>, preservesPartitioning=False)
```

The `map()` transformation is the most basic of all transformations. It evaluates a named or anonymous function for each element within a dataset partition. One

or many `map()` functions can run asynchronously because they shouldn't produce any side effects, maintain state, or attempt to communicate or synchronize with other `map()` operations. That is, they are *shared nothing* operations.

The `preservesPartitioning` argument is an optional Boolean argument intended for use with RDDs with a partitioner defined—typically a key/value pair RDD (as discussed later in this chapter) in which a key is defined and grouped by a key hash or key range. If this parameter is set to `True`, the partitions stay intact. This parameter can be used by the Spark scheduler to optimize subsequent operations, such as joins based on the partitioned key.

Consider Figure 4.7, where the `map()` transformation evaluates a function for each input record and emits a transformed output record. In this case, the `split` function takes a string and produces a list, and each string element in the input data maps to a list element in the output. The result, in this case, is a list of lists.

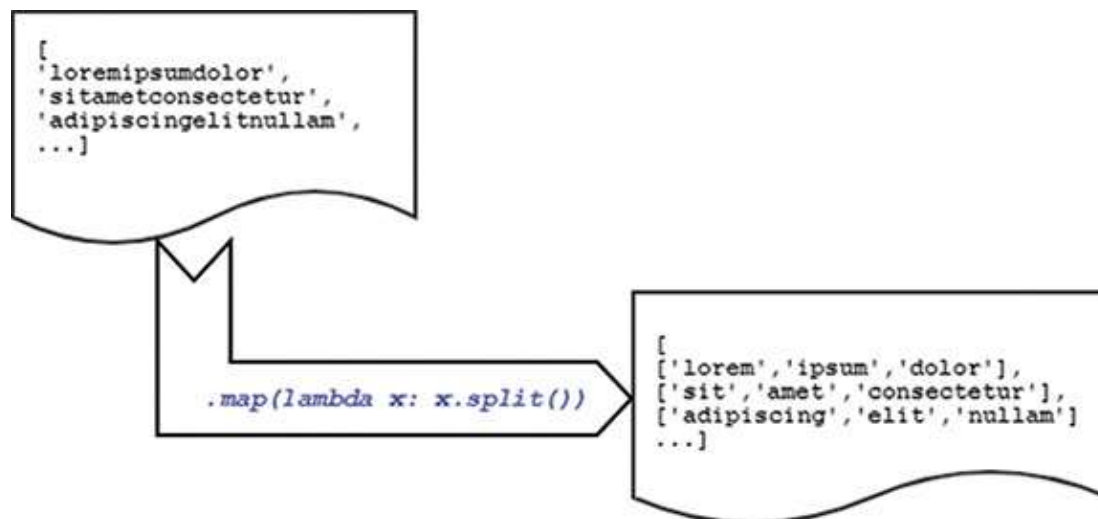


Figure 4.7 The `map()` transformation.

`flatMap()`

Syntax:

[Click here to view code image](#)

```
RDD.flatMap(<function>, preservesPartitioning=False)
```

The `flatMap()` transformation is similar to the `map()` transformation in that it runs a function against each record in the input dataset. However, `flatMap()` “flattens” the output, meaning it removes a level of nesting. For

example, given a list containing lists of strings, flattening would result in a single list of strings—“flattening” all of the nested lists. [Figure 4.8](#) shows the effect of a `flatMap()` transformation using the same anonymous (`lambda`) function as the `map()` operation shown in [Figure 4.7](#). Notice that instead of each string producing a respective list object, all elements are flattened into one list. In other words, `flatMap()`, in this case, produces one combined list as output, in contrast to the list of lists in the `map()` example.

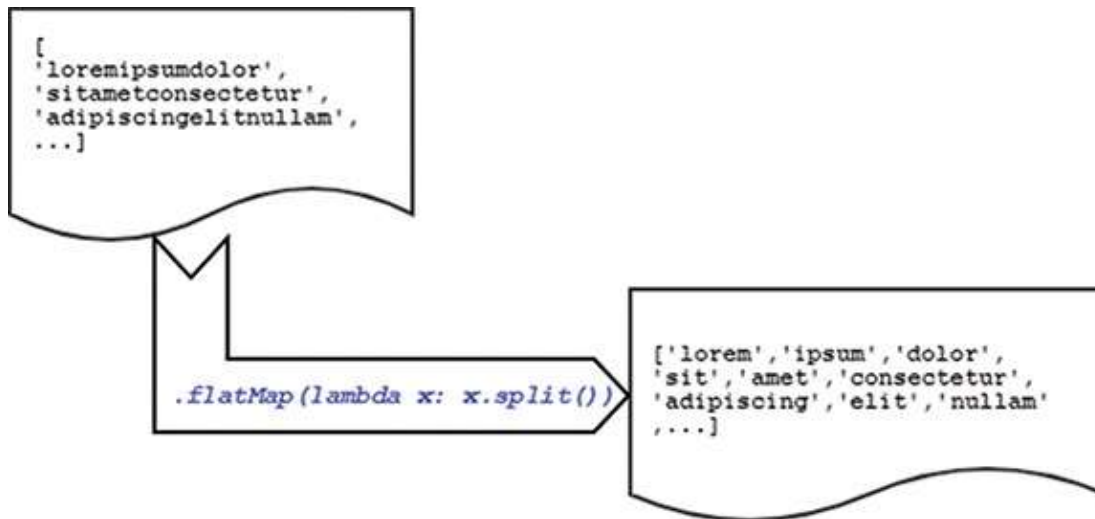


Figure 4.8 The `flatMap()` transformation.

The `preservesPartitioning` argument works the same in `flatMap()` as it does in the `map()` function.

filter()

Syntax:

[Click here to view code image](#)

```
RDD.filter(<function>)
```

The `filter` transformation evaluates a Boolean expression, usually expressed as an anonymous function, against each element in the dataset. The Boolean value returned determines whether the record is included in the resultant output RDD. This is another common transformation used to remove from RDD records that are not required for intermediate processing and that are not included in the final output.

[Listing 4.13](#) shows an example of using the `map()`, `flatMap()`, and

`filter()` transformations together to convert input text to uppercase. It uses `map()` and `flatMap()` to split the text into a combined list of words and then uses `filter()` to filter the list to return only words that are greater than four characters long.

Listing 4.13 The `map()`, `flatMap()`, and `filter()` Transformations

[Click here to view code image](#)

```
licenses = sc.textFile('file:///opt/spark/licenses')
words = licenses.flatMap(lambda x: x.split(' '))
words.take(5)
# returns [u'The', u'MIT', u'License', u'(MIT)', u'']
lowercase = words.map(lambda x: x.lower())
lowercase.take(5)
# returns [u'the', u'mit', u'license', u'(mit)', u'']
longwords = lowercase.filter(lambda x: len(x) > 12)
longwords.take(2)
# returns [u'documentation', u'merchantability,']
```

There is a standard axiom in the world of Big Data programming: “Filter early, filter often.” This refers to the fact that there is no value in carrying records or fields through a process where they are not needed. Both the `filter()` and `map()` functions can be used to achieve this objective. That said, in many cases Spark—through its key runtime characteristic of lazy execution—attempts to optimize routines for you even if you do not explicitly do this yourself.

`distinct()`

Syntax:

[Click here to view code image](#)

```
RDD.distinct(numPartitions=None)
```

The `distinct()` transformation returns a new RDD containing distinct elements from the input RDD. It is used to remove duplicates, where *duplicates* are defined as having all elements or fields within a record that are the same as other records in the dataset. The `numPartitions` argument can redistribute data to a target number of partitions; if this is not supplied or is left at the

default, the number of partitions returned by the `distinct()` transformation is identical to the number of partitions from the RDD operated against.

[Listing 4.14](#) demonstrates the use of the `distinct()` function.

Listing 4.14 The `distinct()` Transformation

[Click here to view code image](#)

```
licenses = sc.textFile('file:///opt/spark/licenses')
words = licenses.flatMap(lambda x: x.split(' '))
lowercase = words.map(lambda x: x.lower())
allwords = lowercase.count()
distinctwords = lowercase.distinct().count()
print "Total words: %s, Distinct Words: %s" % (allwords, distinctwords)
# returns "Total words: 11484, Distinct Words: 892"
```

`groupBy()`

Syntax:

[Click here to view code image](#)

```
RDD.groupBy(<function>, numPartitions=None)
```

The `groupBy()` transformation returns an RDD of items grouped by a specified function. The `<function>` argument is an anonymous or named function used to nominate a key by which to group all elements or to specify an expression to evaluate against elements to determine a group, such as when grouping elements by odd or even numbers of a numeric field in the data.

You can use the `numPartitions` argument to create a specified number of partitions automatically by computing hashes of the key space from the output of the grouping function. For instance, if you want to group an RDD by the days in a week and process each day separately, specify `numPartitions=7`. You will see `numPartitions` specified in numerous Spark transformations, where its behavior is analogous.

[Listing 4.15](#) demonstrates the use of the `groupBy()` function. Notice that `groupBy()` returns an *iterable* object; we will look at how to handle this type of object later in this chapter.

Listing 4.15 Grouping Data in Spark by Using the `groupBy()` Function

[Click here to view code image](#)

```
licenses = sc.textFile('file:///opt/spark/licenses')
words = licenses.flatMap(lambda x: x.split(' ')) \
                 .filter(lambda x: len(x) > 0)
groupedbyfirstletter = words.groupBy(lambda x: x[0].lower())
groupedbyfirstletter.take(1)
# returns:
# [('l', <pyspark.resultiterable.ResultIterable object at
# 0x7f678e9cca20>)]
```

Consider Other Functions for Grouping Data

If your ultimate intention in using `groupBy()` is to aggregate values, such as when performing a `sum()` or `count()` operation, you should opt for more efficient operators for this purpose in Spark, including `aggregateByKey()` and `reduceByKey()`, which we will discuss shortly. The `groupBy()` transformation does not perform any aggregation prior to shuffling data, resulting in more data being shuffled. Furthermore, `groupBy()` requires that all values for a given key fit into memory. The `groupBy()` transformation is useful in some cases, but you should consider these factors before deciding to use this function.

`sortBy()`

Syntax:

[Click here to view code image](#)

```
RDD.sortBy(<keyfunc>, ascending=True, numPartitions=None)
```

The `sortBy()` transformation sorts an RDD by the `<keyfunc>` argument (a named or anonymous function) that nominates the key for a given dataset. It sorts according to the sort order of the key object type. For instance, `int` and `double` data types are sorted numerically, whereas `String` types are sorted in lexicographical order.

The `ascending` argument is a Boolean argument that defaults to `True` and specifies the sort order to be used. A descending sort order is specified by setting

ascending=False.

An example of the `sortBy()` function is shown in [Listing 4.16](#).

Listing 4.16 Sorting Data by Using the `sortBy()` Function

[Click here to view code image](#)

```
readme = sc.textFile('file:///opt/spark/README.md')
words = readme.flatMap(lambda x: x.split(' ')) \
               .filter(lambda x: len(x) > 0)
sortByfirstletter = words.sortBy(lambda x: x[0].lower(),
                                  ascending=False)
sortByfirstletter.take(5)
# returns ['You', 'you', 'You', 'you', 'you']
```

Basic RDD Actions

Recall that actions in Spark either return values, as is the case with `count()`; return data, as is the case with `collect()`; or save data externally, as is the case with `saveAsTextFile()`. In all cases, actions force computation of an RDD and all of its parents. Some actions return either a count, an aggregation of the data, or part or all of the data in an RDD. In contrast, `foreach()` is an action that performs a *function* on each element of an RDD. The following sections look at some of the basic actions in the core Spark API.

`count()`

Syntax:

```
RDD.count()
```

The `count()` action takes no arguments and returns a `long` value, which represents the count of the elements in the RDD. [Listing 4.17](#) shows a simple `count()` example. Note that with actions that take no arguments, you need to include empty parentheses, `()`, after the action name.

Listing 4.17 The `count()` Action

[Click here to view code image](#)

```
licenses = sc.textFile('file:///opt/spark/licenses')
words = licenses.flatMap(lambda x: x.split(' '))
words.count()
# returns 11484
```

collect()

Syntax:

```
RDD.collect()
```

The `collect()` action returns a list that contains all the elements in an RDD to the Spark Driver. Because `collect()` does not restrict the output, which can be quite large and can potentially cause out-of-memory errors on the Driver, it is typically useful for only small RDDs or development. [Listing 4.18](#) demonstrates the `collect()` action.

Listing 4.18 The `collect()` Action

[Click here to view code image](#)

```
licenses = sc.textFile('file:///opt/spark/licenses')
words = licenses.flatMap(lambda x: x.split(' '))
words.collect()
# returns [u'The', u'MIT', u'License', u'(MIT)', u'', u'Copyright', ...]
```

take()

Syntax:

```
RDD.take(n)
```

The `take()` action returns the first `n` elements of an RDD. The elements taken are not in any particular order; in fact, the elements returned from a `take()` action are non-deterministic, meaning they can differ if the same action is run again, particularly in a fully distributed environment. There is a similar Spark function, `takeOrdered()`, which takes the first `n` elements ordered based on a key supplied by a key function.

For RDDs that span more than one partition, `take()` scans one partition and uses the results from that partition to estimate the number of additional partitions

needed to satisfy the number requested.

[Listing 4.19](#) shows an example of the `take()` action.

Listing 4.19 The `take()` Action

[Click here to view code image](#)

```
licenses = sc.textFile('file:///opt/spark/licenses')
words = licenses.flatMap(lambda x: x.split(' '))
words.take(3)
# returns [u'The', u'MIT', u'License']
```

`top()`

Syntax:

```
RDD.top(n, key=None)
```

The `top()` action returns the top `n` elements from an RDD, but unlike with `take()`, with `top()` the elements are ordered and returned in descending order. Order is determined by the object type, such as numeric order for integers or dictionary order for strings.

The `key` argument specifies the key by which to order the results to return the top `n` elements. This is an optional argument; if it is not supplied, the key will be inferred from the elements in the RDD.

[Listing 4.20](#) shows the top three distinct words sorted from a text file in descending lexicographical order.

Listing 4.20 The `top()` Action

[Click here to view code image](#)

```
readme = sc.textFile('file:///opt/spark/README.md')
words = readme.flatMap(lambda x: x.split(' '))
words.distinct().top(3)
# returns [u'your', u'you', u'with']
```

`first()`

Syntax:

```
RDD.first()
```

The `first()` action returns the first element in this RDD. Similar to the `take()` and `collect()` actions and unlike the `top()` action, `first()` does not consider the order of elements and is a non-deterministic operation, especially in fully distributed environments.

As you can see from [Listing 4.21](#), the primary difference between `first()` and `take(1)` is that `first()` returns an atomic data element, and `take()` (even if `n = 1`) returns a list of data elements. The `first()` action is useful for inspecting the output of an RDD as part of development or data exploration.

Listing 4.21 The `first()` Action

[Click here to view code image](#)

```
readme = sc.textFile('file:///opt/spark/README.md')
words = readme.flatMap(lambda x: x.split(' ')) \
    .filter(lambda x: len(x) > 0)
words.distinct().first()
# returns a string: u'project.'
words.distinct().take(1)
# returns a list with one string element: [u'project.']
```

The `reduce()` and `fold()` actions are aggregate actions, each of which executes a commutative and/or an associative operation, such as summing a list of values, against an RDD. *Commutative* and *associative* are the operative terms here. This makes the operations independent of the order in which they run, and this is integral to distributed processing because the order isn't guaranteed. Here is the general form of the commutative characteristics:

$$x + y = y + x$$

And here is the general form of the associative characteristics:

$$(x + y) + z = x + (y + z)$$

The following sections look at the main Spark actions that perform aggregations.

`reduce()`

Syntax:

```
RDD.reduce(<function>)
```

The `reduce()` action reduces the elements of an RDD using a specified commutative and/or associative operator. The `<function>` argument specifies two inputs (`lambda x, y: ...`) that represent values in a sequence from the specified RDD. [Listing 4.22](#) shows an example of a `reduce()` operation to produce a sum against a list of numbers.

Listing 4.22 Summing Values in an RDD by Using the `reduce()` Action

[Click here to view code image](#)

```
numbers = sc.parallelize([1,2,3,4,5,6,7,8,9])
numbers.reduce(lambda x, y: x + y)
# returns 45
```

`fold()`

Syntax:

```
RDD.fold(zeroValue, <function>)
```

The `fold()` action aggregates the elements of each partition of an RDD and then performs the aggregate operation against the results for all, using a given function and a `zeroValue`. Although `reduce()` and `fold()` are similar in function, they differ in that `fold()` is not commutative, and thus an initial and final value (`zeroValue`) is required. A simple example is a `fold()` action with `zeroValue=0`, as shown in [Listing 4.23](#).

Listing 4.23 The `fold()` Action

[Click here to view code image](#)

```
numbers = sc.parallelize([1,2,3,4,5,6,7,8,9])
numbers.fold(0, lambda x, y: x + y)
# returns 45
```

The `fold()` action in [Listing 4.23](#) looks exactly the same as the `reduce()`

action in [Listing 4.22](#). However, [Listing 4.24](#) demonstrates a clear functional difference in the two actions. The `fold()` action provides a `zeroValue` that is added to the beginning and end of the function supplied as input to the `fold()` action, generalized here:

[Click here to view code image](#)

```
result = zeroValue + ( 1 + 2 ) + 3 . . . + zeroValue
```

This allows `fold()` to operate on an empty RDD, whereas `reduce()` produces an exception with an empty RDD.

Listing 4.24 The `fold()` Action Compared with `reduce()`

[Click here to view code image](#)

```
empty = sc.parallelize([])
empty.reduce(lambda x, y: x + y)
# returns:
# ValueError: Cannot reduce() empty RDD
empty.fold(0, lambda x, y: x + y)
# returns 0
```

There is also a similar `aggregate()` action in the Spark RDD API.

`foreach()`

Syntax:

```
RDD.foreach(<function>)
```

The `foreach()` action applies a function specified in the `<function>` argument, anonymous or named, to all elements of an RDD. Because `foreach()` is an action rather than a transformation, you can perform functions otherwise not possible or intended in transformations, such as a `print()` function. Although Python `lambda` functions don't allow you to execute a `print()` statement directly, you can use a named function that executes `print()` instead. [Listing 4.25](#) shows an example of this.

Listing 4.25 The `foreach()` Action

[Click here to view code image](#)

```
def printfunc(x):
    print(x)
licenses = sc.textFile('file:///opt/spark/licenses')
longwords = licenses.flatMap(lambda x: x.split(' ')) \
    .filter(lambda x: len(x) > 12)
longwords.foreach(lambda x: printfunc(x))
# returns:
# ...
# Redistributions
# documentation
# distribution.
# MERCHANTABILITY
# ...
```

Transformations on PairRDDs

Key/value pair RDDs, or simply PairRDDs, contain records consisting of keys and values. The keys can be simple objects such as integer or string objects or complex objects such as tuples. The values can range from scalar values to data structures such as lists, tuples, dictionaries, or sets. This is a common data representation in multi-structured data analysis on schema-on-read and NoSQL systems. PairRDDs and their constituent functions are integral to functional Spark programming. These functions are broadly classified into four categories:

- Dictionary functions
- Functional transformations
- Grouping, aggregation, and sorting operations
- Join functions, which we discuss specifically in the next section

Dictionary functions return a set of keys or values from a key/value pair RDD. Examples include `keys()` and `values()`.

Earlier in this chapter we looked at other aggregate operations, including `reduce()` and `fold()`. These are conceptually similar in that they aggregate values in an RDD based on a key, but there is a fundamental difference: `reduce()` and `fold()` are *actions*, which means they force computation and produce a result, whereas `reduceByKey()` and `foldByKey()`, which we discuss shortly, are *transformations*, meaning they are lazily evaluated and return

a new RDD.

keys()

Syntax:

```
RDD.keys()
```

The `keys()` function returns an RDD with the keys from a key/value pair RDD or the first element from each tuple in a key/value pair RDD. [Listing 4.26](#) demonstrates using the `keys()` function.

Listing 4.26 The `keys()` Function

[Click here to view code image](#)

```
kvpairs = sc.parallelize([('city', 'Hayward')
                          , ('state', 'CA')
                          , ('zip', 94541)
                          , ('country', 'USA')])
kvpairs.keys().collect()
# returns ['city', 'state', 'zip', 'country']
```

values()

Syntax:

```
RDD.values()
```

The `values()` function returns an RDD with values from a key/value pair RDD or the second element from each tuple in a key/value pair RDD. [Listing 4.27](#) demonstrates using the `values()` function.

Listing 4.27 The `values()` Function

[Click here to view code image](#)

```
kvpairs = sc.parallelize([('city', 'Hayward')
                          , ('state', 'CA')
                          , ('zip', 94541)
                          , ('country', 'USA')])
kvpairs.values().collect()
```

```
# returns ['Hayward', 'CA', 94541, 'USA']
```

keyBy()

Syntax:

```
RDD.keyBy(<function>)
```

The `keyBy()` transformation creates a tuple consisting of a key and a value from the elements in the RDD by applying a function specified by the `<function>` argument. The value is the complete tuple from which the key was derived.

Consider a list of locations as tuples with a schema of `city`, `country`, `location_no`. Say that you want the `location_no` field to be your key. The example in [Listing 4.28](#) demonstrates the use of the `keyBy()` function to create new tuples in which the first element is the key and the second element, the value, is a tuple containing all fields from the original tuple.

Listing 4.28 The `keyBy()` Transformation

[Click here to view code image](#)

```
locations = sc.parallelize([('Hayward', 'USA', 1)
                           , ('Baumholder', 'Germany', 2)
                           , ('Alexandria', 'USA', 3)
                           , ('Melbourne', 'Australia', 4)])
bylocno = locations.keyBy(lambda x: x[2])
bylocno.collect()
# returns:
#[(1, ('Hayward', 'USA', 1)), (2, ('Baumholder', 'Germany', 2)),
# (3, ('Alexandria', 'USA', 3)), (4, ('Melbourne', 'Australia', 4))]
```

Recall that `x[2]` in [Listing 4.28](#) refers to the third element in list `x`, as Python list elements are ordinal numbers, starting with 0.

Functional transformations available for key/value pair RDDs work similarly to the more general functional transformations you learned about earlier. The difference is that these functions operate specifically on either the key or value element within a tuple—the key/value pair, in this case. Functional transformations include `mapValues()` and `flatMapValues()`.

mapValues()

Syntax:

```
RDD.mapValues(<function>)
```

The `mapValues()` transformation passes each value in a key/value pair RDD through a function (a named or anonymous function specified by the `<function>` argument) without changing the keys. Like its generalized equivalent `map()`, `mapValues()` outputs one element for each input element.

The original RDD's partitioning is not affected.

flatMapValues()

Syntax:

```
RDD.flatMapValues(<function>)
```

The `flatMapValues()` transformation passes each value in a key/value pair RDD through a function without changing the keys and produces a flattened list. It works exactly like `flatMap()`, which we looked at earlier, returning zero to many output elements per input element.

Much as with `mapValues()`, with `flatMapValues()` the original RDD's partitioning is retained.

The easiest way to contrast `mapValues()` and `flatMapValues()` is to look at a practical example. Consider a text file containing a city and a pipe-delimited list of temperatures, as shown here:

[Click here to view code image](#)

```
Hayward, 71|69|71|71|72  
Baumholder, 46|42|40|37|39  
Alexandria, 50|48|51|53|44  
Melbourne, 88|101|85|77|74
```

[Listing 4.29](#) simulates the loading of this data into an RDD and uses `mapValues()` to create a list of key/value pair tuples containing the city and a list of temperatures for the city. It shows the use of `flatMapValues()` with the same function against the same RDD to create tuples containing the city and a number for each temperature recorded for the city.

A simple way to describe this is that `mapValues()` creates one element per city containing the city name and a list of five temperatures for the city, whereas

`flatMapValues()` *flattens* the lists to create five elements per city with the city name and a temperature value.

Listing 4.29 The `mapValues()` and `flatMapValues()` Transformations

[Click here to view code image](#)

```
locwtemps = sc.parallelize(['Hayward,71|69|71|71|72',
                            'Baumholder,46|42|40|37|39',
                            'Alexandria,50|48|51|53|44',
                            'Melbourne,88|101|85|77|74'])

kvpairs = locwtemps.map(lambda x: x.split(','))
kvpairs.take(4)
# returns :
# [['Hayward', '71|69|71|71|72'],
#  ['Baumholder', '46|42|40|37|39'],
#  ['Alexandria', '50|48|51|53|44'],
#  ['Melbourne', '88|101|85|77|74']]

locwtemplist = kvpairs.mapValues(lambda x: x.split('|')) \
                    .mapValues(lambda x: [int(s) for s in x])

locwtemplist.take(4)
# returns :
# [('Hayward', [71, 69, 71, 71, 72]),
#  ('Baumholder', [46, 42, 40, 37, 39]),
#  ('Alexandria', [50, 48, 51, 53, 44]),
#  ('Melbourne', [88, 101, 85, 77, 74])]

locwtemps = kvpairs.flatMapValues(lambda x: x.split('|')) \
                .map(lambda x: (x[0], int(x[1])))

locwtemps.take(3)
# returns :
# [('Hayward', 71), ('Hayward', 69), ('Hayward', 71)]
```

Grouping, aggregation, and sorting operations are functionally analogous to their more generalized forms discussed earlier in this chapter (`groupBy()` and `sortBy()`), again with the difference being that these functions operate specifically on RDDs composed of key/value pairs.

Be Cautious of the Repartitioning and Shuffling Effects of

Some Functions

Be aware that some functions, such as `groupByKey()` and `reduceByKey()`, may result in a repartitioning or require shuffling. Shuffling is a relatively expensive operation because it requires the movement of data between Spark Executors, often located on different Worker nodes. These operations are often necessary and unavoidable, but in some cases, by understanding the planning and execution of an RDD's lineage, you may be able to optimize these operations. We discuss partitioning in more detail in [Chapter 5](#).

`groupByKey()`

Syntax:

[Click here to view code image](#)

```
RDD.groupByKey(numPartitions=None, partitionFunc=<hash_fn>)
```

The `groupByKey()` transformation groups the values for each key in a key/value pair RDD into a single sequence.

The `numPartitions` argument specifies how many partitions—how many groups, that is—to create. The partitions are created using the `partitionFunc` argument, which defaults to Spark's built-in hash partitioner. If `numPartitions` is `None`, which is the default, then the configured system default number of partitions is used (`spark.default.parallelism`).

Consider the output from [Listing 4.29](#). If you want to calculate the average temperature by city, you first need to group all the values together by their city and then compute the averages. [Listing 4.30](#) shows how to use `groupByKey()` to accomplish this.

Listing 4.30 The `groupByKey()` Transformation

[Click here to view code image](#)

```
# continued from Listing 4.29
grouped = locwtemps.groupByKey()
grouped.take(1)
# returns:
# [('Melbourne', <pyspark.resultiterable.ResultIterable object at
0x7f121ce11390>)]
```

```
avgtemps = grouped.mapValues(lambda x: sum(x)/len(x))
avgtemps.collect()
# returns:
# [('Melbourne', 85), ('Baumholder', 40), ('Alexandria', 49),
# ('Hayward', 70)]
```

Notice that `groupByKey()` returns a `resultiterable` object for the grouped values. An *iterable* object in Python is a sequence object that can loop over. Many functions in Python accept iterables as input, such as the `sum()` and `len()` functions.

Consider Using `reduceByKey()` or `foldByKey()` Instead of `groupByKey()`

If you group values for the purposes of aggregation, such as by using a `sum()` or `count()` for each key, then using `reduceByKey()` or `foldByKey()` provides much better performance in many cases. This is because the results of the aggregation function are combined before the shuffle, resulting in a reduced amount of data being shuffled.

`reduceByKey()`

Syntax:

[Click here to view code image](#)

```
RDD.reduceByKey(<function>, numPartitions=None, partitionFunc=<hash_fn>)
```

The `reduceByKey()` transformation merges the values for the keys by using an associative function. The `reduceByKey()` method is called on a dataset of key/value pairs and returns a dataset of key/value pairs, aggregating values for each key. This function is expressed as follows:

$$v_n, v_{n+1} \Rightarrow v_{result}$$

The `numPartitions` and `partitionFunc` arguments behave exactly the same as in the `groupByKey()` function. The `numPartitions` value is effectively the number of reduce tasks to execute, and you can increase this to obtain a higher degree of parallelism. The `numPartitions` value also affects the number of files produced with `saveAsTextFile()` or other file-producing Spark actions. For instance, `numPartitions=2` produces two

output files when the RDD saves to disk.

[Listing 4.31](#) takes the same input key/value pairs and produces the same results (average temperatures per city) as the previous `groupByKey()` example—but using the `reduceByKey()` function instead. This method is preferred for reasons we will discuss shortly.

Listing 4.31 Using the `reduceByKey()` Function to Average Values by Key

[Click here to view code image](#)

```
# continued from Listing 4.29
temptups = locwtemps.mapValues(lambda x: (x, 1))
# creates tuples (city, (temp, 1))
inputstoavg = temptups.reduceByKey(lambda x, y: (x[0]+y[0], x[1]+y[1]))
# sums temperatures by city
averages = inputstoavg.map(lambda x: (x[0], x[1][0]/x[1][1]))
# divides the sum of temperatures by key by the number of readings
averages.take(4)
# returns :
# [('Baumholder', 40.8),
#  ('Melbourne', 85.0),
#  ('Alexandria', 49.2),
#  ('Hayward', 70.8)]
```

Averaging is not an associative operation; you can get around this by creating tuples containing the sum total of values for each key and the count for each key—operations that are associative and commutative—and then computing the average as a final step, as shown in [Listing 4.31](#).

Note that `reduceByKey()` is efficient because it combines values locally on each Executor before each of the combined lists sends to a remote Executor or Executors running the final reduce stage. This is a shuffle operation.

Because the same associative and commutative function are run on the local Executor or Worker and again on a remote Executor or Executors, taking a sum function, for example, you can think of this as adding a list of sums as opposed to summing a bigger list of individual values. Because there is less data sent in the shuffle phase, `reduceByKey()` using a sum function generally performs

better than `groupByKey()` followed by a `sum()` function.

foldByKey()

Syntax:

[Click here to view code image](#)

```
RDD.foldByKey(zeroValue, <function>, numPartitions=None,  
partitionFunc=<hash_fn>)
```

The `foldByKey()` transformation is functionally similar to the `fold()` action discussed in the previous section. However, `foldByKey()` is a transformation that works with predefined key/value pair elements (see [Listing 4.32](#)). Both `foldByKey()` and `fold()` provide a `zeroValue` argument of the same type to be used if the RDD is empty.

The function supplied is in the generalized aggregate function form:

$$v_n, v_{n+1} \Rightarrow v_{result}$$

This is the same generalization used by the `reduceByKey()` transformation.

The `numPartitions` and the `partitionFunc` arguments have the same effect as they do with the `groupByKey()` and `reduceByKey()` transformations.

Listing 4.32 A foldByKey() Example to Find Maximum Value by Key

[Click here to view code image](#)

```
#continued from Listing 4.29  
maxbycity = locwtemps.foldByKey(0, lambda x, y: x if x > y else y)  
maxbycity.collect()  
# returns :  
# [('Baumholder', 46), ('Melbourne', 101), ('Alexandria', 53),  
('Hayward', 72)]
```

There is also a similar method called `aggregateByKey()` in the Spark RDD API.

sortByKey()

Syntax:

[Click here to view code image](#)

```
RDD.sortByKey(ascending=True, numPartitions=None, keyfunc=<function>)
```

The `sortByKey()` transformation sorts a key/value pair RDD by the predefined key. The sort order is dependent on the underlying key object type, where numeric types are sorted numerically and so on. The difference between `sort()`, discussed earlier, and `sortByKey()` is that `sort()` requires you to identify the key by which to sort, whereas `sortByKey()` is aware of the key already.

Keys are sorted in the order provided by the `ascending` argument, which defaults to `True`. The `numPartitions` argument specifies how many resultant partitions to output using a range partitioning function. The `keyfunc` argument is an optional parameter to use if you want to derive a key from passing the predefined key through another function, as in this example:

[Click here to view code image](#)

```
keyfunc=lambda k: k.lower()
```

[Listing 4.33](#) shows the use of the `sortByKey()` transformation. The first example shows a simple sort based on the key: a string representing the city name, sorted alphabetically. In the second example, the keys and values are inverted to make the temperature the key and then use `sortByKey()` to list the temperatures in descending numeric order, with the highest temperatures first.

Listing 4.33 The `sortByKey()` Transformation

[Click here to view code image](#)

```
# continued from Listing 4.29
sortedbykey = locwtemps.sortByKey()
sortedbykey.take(4)
# returns:
# [('Alexandria', 50), ('Alexandria', 48), ('Alexandria', 51),
  ('Alexandria', 53)]
sortedbyval = locwtemps.map(lambda x: (x[1],x[0])) \
                        .sortByKey(ascending=False)
sortedbyval.take(4)
# returns:
```

```
# [(101, 'Melbourne'), (88, 'Melbourne'), (85, 'Melbourne'), (77, 'Melbourne')]
```

MapReduce and Word Count Exercise

MapReduce is a platform- and language-independent programming model or design pattern at the heart of most Big Data and NoSQL platforms. Although many abstractions of MapReduce exist, such as Pig and Hive, which allow you to process data without explicitly implementing map or reduce functions, understanding the concepts behind MapReduce is fundamental to truly understanding distributed programming and data processing in Spark.

Word Count, a sample program often referred to as the “Hello World” of MapReduce, is a simple algorithm often used to represent and demonstrate the MapReduce programming model. If you have previously read any Hadoop or Spark training material or tutorials, you are probably tired of seeing Word Count examples, or you may be scratching your head, trying to understand the fixation with counting words.

Word Count is the most prevalent example used when describing the MapReduce programming model because it is easy to understand and demonstrates all the components of the MapReduce model. Many real-life problems solved with MapReduce are simply adaptations or derivations of Word Count (for instance, counting occurrences of events in a large corpus of log files, or text mining functions such as *TF-IDF* [*Term Frequency-Inverse Document Frequency*]). When you understand Word Count, you understand MapReduce, and the problem-solving possibilities are endless. Let’s walk through a simple example using Spark now:

1. Using your single-node Spark installation, download the `shakespeare.txt` file (works of Shakespeare) from this link:

[Click here to view code image](#)

<https://s3.amazonaws.com/sparkusingpython/shakespeare/shakespeare.txt>

You can use `wget` or `curl` to download this file.

2. Place the file in the `/opt/spark/data` directory of your Spark installation:

[Click here to view code image](#)

```
$ sudo mv shakespeare.txt /opt/spark/data
```


Note that if you have HDFS available to you (for example, with AWS EMR, Databricks, or a Hadoop distribution that includes Spark), you can upload the file to HDFS and use it as an alternative.

3. Open a PySpark shell in local mode:

[Click here to view code image](#)

```
$ pyspark --master local
```

If you have a Hadoop cluster or distributed Spark Standalone cluster accessible, you are free to use it instead by specifying one of the following:

[Click here to view code image](#)

```
--master yarn  
--master spark://<yoursparkmaster>:7077
```

Note that if your Python binary is not `python` (for instance, it may be `py` or `python3` depending upon your release), you need to direct Spark to the correct file. This can be done using the following environment variable settings:

[Click here to view code image](#)

```
$ export PYSPARK_PYTHON=python3  
$ export PYSPARK_DRIVER_PYTHON=python3
```

4. From your PySpark session, import the Python `re` (Regular Expression) module, which you will use to tokenize the file:

```
import re
```

5. Load the `shakespeare.txt` file into an RDD named `doc`:

[Click here to view code image](#)

```
doc = sc.textFile("file:///opt/spark/data/shakespeare.txt")
```

6. Filter empty lines from the RDD, split lines by whitespace, and flatten the lists of words into one list:

[Click here to view code image](#)

```
flattened = doc.filter(lambda line: len(line) > 0) \  
    .flatMap(lambda line: re.split('\W+', line))
```

7. Inspect the `flattened` RDD:

```
flattened.take(6)
```

8. Map text to lowercase, remove empty strings, and then convert to key/value pairs in the form `(word, 1)`:

[Click here to view code image](#)

```
kvpairs = flattened.filter(lambda word: len(word) > 0) \
    .map(lambda word: (word.lower(), 1))
```

9. Inspect the `kvpairs` RDD. Notice that the RDD created is a *PairRDD* representing a collection of key/value pairs:

```
kvpairs.take(5)
```

10. Count each word and sort results in reverse alphabetic order:

[Click here to view code image](#)

```
countsbyword = kvpairs.reduceByKey(lambda v1, v2: v1 + v2) \
    .sortByKey(ascending=False)
```

11. Inspect the `countsbyword` RDD:

```
countsbyword.take(5)
```

12. Find the top five most-used words:

[Click here to view code image](#)

```
# invert the kv pair to make the count the key and sort
topwords = countsbyword.map(lambda x: (x[1], x[0])) \
    .sortByKey(ascending=False)
```

13. Inspect the `topwords` RDD:

```
topwords.take(5)
```

Note how the `map()` function is used in step 12 to invert the key and value. This is a common approach to performing an operation known as a *secondary sort*, which is a means to sort values that are not sorted by default.

Now exit your `pyspark` session by pressing `Ctrl+D`.

14. Now put it all together and run it as a complete Python program by using `spark-submit`. First, minimize the amount of logging by creating and configuring a `log4j.properties` file in the `conf` directory of your Spark installation. Do this by executing the following command from a Linux terminal (or an analogous operation if you are using another operating system):

[Click here to view code image](#)

```
sed \
"s/log4j.rootCategory=INFO, console/log4j.rootCategory=ERROR, \
console/" \
$SPARK_HOME/conf/log4j.properties.template \
> $SPARK_HOME/conf/log4j.properties
```

15. Create a new file named `wordcounts.py` and add the following code to the file:

[Click here to view code image](#)

```
import sys, re
from pyspark import SparkConf, SparkContext
conf = SparkConf().setAppName('Word Counts')
sc = SparkContext(conf=conf)

# check command line arguments
if (len(sys.argv) != 3):
    print("""\
This program will count occurrences of each word in a document or
documents
and return the counts sorted by the most frequently occurring words

Usage: wordcounts.py <input_file_or_dir> <output_dir>
""")
    sys.exit(0)
else:
    inputpath = sys.argv[1]
    outputdir = sys.argv[2]

# count and sort word occurrences
wordcounts = sc.textFile("file://" + inputpath) \
    .filter(lambda line: len(line) > 0) \
    .flatMap(lambda line: re.split('\W+', line)) \
    .filter(lambda word: len(word) > 0) \
    .map(lambda word: (word.lower(), 1)) \
    .reduceByKey(lambda v1, v2: v1 + v2) \
    .map(lambda x: (x[1], x[0])) \
    .sortByKey(ascending=False) \
    .persist()
wordcounts.saveAsTextFile("file://" + outputdir)
top5words = wordcounts.take(5)
justwords = []
for wordsandcounts in top5words:
    justwords.append(wordsandcounts[1])
print("The top five words are : " + str(justwords))
print("Check the complete output in " + outputdir)
```

16. Execute your program by using the following command:

[Click here to view code image](#)

```
$ spark-submit --master local \
wordcounts.py \
$SPARK_HOME/data/shakespeare.txt \
$SPARK_HOME/data/wordcounts
```

You should see the top five words displayed in the console. Check the output directory `$SPARK_HOME/data/wordcounts`; you should see one file in this directory (`part-00000`) because you used only one partition for this exercise. If you used more than one partition, you would see additional files (`part-00001`, `part-00002`, and so on). Open the file and inspect the contents.

17. Run the command from step 16 again. It should fail because the `wordcounts` directory already exists and cannot be overwritten. Simply remove or rename this directory or change the output directory for the next operation to a directory that does not exist, such as `wordcounts2`.

The complete source code for this exercise can be found in the `wordcount` folder at https://github.com/sparktraining/spark_using_python.

Join Transformations

Join operations are analogous to the `JOIN` operations you routinely see in SQL programming. Join functions combine records from two RDDs based on a common field, a key. Because join functions in Spark require a key to be defined, they operate on key/value pair RDDs.

The following is a quick refresher on joins—which you may want to skip if you have a relational database background:

- A *join* operates on two different datasets, where one field in each dataset is nominated as a key (a *join key*). The datasets are referred to in the order in which they are specified. For instance, the first dataset specified is considered the *left* entity or dataset, and the second dataset specified is considered the *right* entity or dataset.
- An *inner join*, often simply called a *join* (where the “inner” is inferred), returns all elements or records from both datasets, where the nominated key is present in both datasets.
- An *outer join* does not require keys to match in both datasets. Outer joins

are implemented as either a left outer join, a right outer join, or a full outer join.

- A *left outer join* returns all records from the left (or first) dataset along with matched records only (by the specified key) from the right (or second) dataset.
- A *right outer join* returns all records from the right (or second) dataset along with matched records only (by the specified key) from the left (or first) dataset.
- A *full outer join* returns all records from both datasets whether there is a key match or not.

Joins are some of the most commonly required transformations in the Spark API, so it is imperative that you understand these functions and become comfortable using them.

To illustrate the use of the different join types in the Spark RDD API, let's consider a dataset from a fictitious retailer that includes an entity containing stores and an entity containing salespeople, loaded into RDDs, as shown in [Listing 4.34](#).

Listing 4.34 Datasets Used to Demonstrate Join Types in Spark

[Click here to view code image](#)

```
stores = sc.parallelize([(100, 'Boca Raton'),
                        (101, 'Columbia'),
                        (102, 'Cambridge'),
                        (103, 'Naperville')])
# stores schema (store_id, store_location)
salespeople = sc.parallelize([(1, 'Henry', 100),
                             (2, 'Karen', 100),
                             (3, 'Paul', 101),
                             (4, 'Jimmy', 102),
                             (5, 'Janice', None)])
# salespeople schema (salesperson_id, salesperson_name, store_id)
```

The following sections look at the available join transformations in Spark, their usage, and some examples.

join()

Syntax:

[Click here to view code image](#)

```
RDD.join(<otherRDD>, numPartitions=None)
```

The `join()` transformation is an implementation of an inner join, matching two key/value pair RDDs by their key.

The optional `numPartitions` argument determines how many partitions to create in the resultant dataset. If this is not specified, the default value for the `spark.default.parallelism` configuration parameter is used. The `numPartitions` argument has the same behavior for other types of join operations in the Spark API as well.

The RDD returned is a structure containing the matched key and a value that is a tuple containing all the matched records from both RDDs as a list object. (This is where it may sound a bit foreign to you if you are used to performing **INNER JOIN** operations in SQL, which returns a flattened list of columns from both entities.)

[Listing 4.35](#) demonstrates how a `join()` operation works in Spark.

Listing 4.35 The `join()` Transformation

[Click here to view code image](#)

```
salespeople.keyBy(lambda x: x[2]) \
    .join(stores).collect()
# returns: [(100, ((1, 'Henry', 100), 'Boca Raton')),
#           (100, ((2, 'Karen', 100), 'Boca Raton')),
#           (102, ((4, 'Jimmy', 102), 'Cambridge')),
#           (101, ((3, 'Paul', 101), 'Columbia'))]
```

This `join()` operation returns all salespeople assigned to stores keyed by the store ID (the join key) along with the entire store record and salesperson record. Notice that the resultant RDD contains duplicate data. You could (and should in many cases) follow the `join()` with a `map()` transformation to prune fields or project only the fields required for further processing.

Optimizing Joins in Spark

Joins involving RDDs that span more than one partition—and many do—require a shuffle. Spark generally plans and implements this activity to achieve the most optimal performance possible; however, a simple axiom to remember is “join large by small.” This means to reference the large RDD (the one with the most elements, if this is known) first, followed by the smaller of the two RDDs. This will seem strange for users coming from relational database programming backgrounds, but unlike with relational database systems, joins in Spark are relatively inefficient. And unlike with most databases, there are no indexes or statistics to optimize the join, so the optimizations you provide are essential to maximizing performance.

leftOuterJoin()

Syntax:

[Click here to view code image](#)

```
RDD.leftOuterJoin(<otherRDD>, numPartitions=None)
```

The `leftOuterJoin()` transformation returns all elements or records from the first RDD referenced. If keys from the first (or left) RDD are present in the right RDD, then the right RDD record is returned along with the left RDD record. Otherwise, the right RDD record is `None` (empty).

The example shown in [Listing 4.36](#) uses the `leftOuterJoin()` transformation to identify salespeople with no stores.

Listing 4.36 The `leftOuterJoin()` Transformation

[Click here to view code image](#)

```
salespeople.keyBy(lambda x: x[2]) \  
    .leftOuterJoin(stores) \  
    .filter(lambda x: x[1][1] is None) \  
    .map(lambda x: "salesperson " + x[1][0][1] + " has no store") \  
\  
    .collect()  
# returns ['salesperson Janice has no store']
```

rightOuterJoin()

Syntax:

[Click here to view code image](#)

```
RDD.rightOuterJoin(<otherRDD>, numPartitions=None)
```

The `rightOuterJoin()` transformation returns all elements or records from the second RDD referenced. If keys from the second (or right) RDD are present in the left RDD, then the left RDD record is returned along with the right RDD record. Otherwise, the left RDD record is `None` (empty).

[Listing 4.37](#) shows an example of how the `rightOuterJoin()` transformation can be used to identify stores with no salespeople.

Listing 4.37 The `rightOuterJoin()` Transformation

[Click here to view code image](#)

```
salespeople.keyBy(lambda x: x[2]) \
    .rightOuterJoin(stores) \
    .filter(lambda x: x[1][0] is None) \
    .map(lambda x: x[1][1] + " store has no salespeople") \
    .collect()
# returns ['Naperville store has no salespeople']
```

fullOuterJoin()

Syntax:

[Click here to view code image](#)

```
RDD.fullOuterJoin(<otherRDD>, numPartitions=None)
```

The `fullOuterJoin()` transforms all elements from both RDDs whether there is a key matched or not. Keys not matched from either the left or right dataset are represented as `None` (empty).

[Listing 4.38](#) shows an example of how the `fullOuterJoin()` transformation can be used to identify stores with no salespeople *as well as* salespeople with no stores.

Listing 4.38 The `fullOuterJoin()` Transformation

[Click here to view code image](#)

```
salespeople.keyBy(lambda x: x[2]) \
    .fullOuterJoin(stores) \
    .filter(lambda x: x[1][0] is None or x[1][1] is None) \
    .collect()
# returns [(, ([5, 'Janice', ], None)), (103, (None, [103, 'Naperville']))]
```

cogroup()

Syntax:

[Click here to view code image](#)

```
RDD.cogroup(<otherRDD>, numPartitions=None)
```

The `cogroup()` transformation groups multiple key/value pair datasets by a key. It is somewhat similar conceptually to a `fullOuterJoin()`, but there are a few key differences in its implementation:

- The `cogroup()` transformation returns an *iterable* object, similar to the object returned from the `groupByKey()` function you saw earlier.
- The `cogroup()` transformation groups multiple elements from both RDDs into *iterable* objects, whereas `fullOuterJoin()` creates separate output elements for the same key.
- The `cogroup()` transformation can group three or more RDDs using the Scala API or the `groupWith()` function alias.

The resultant RDD output from a `cogroup()` operation of two RDDs (*A*, *B*) with a key *K* could be summarized as:

[Click here to view code image](#)

```
[K, Iterable(K, VA, ...), Iterable(K, VB, ...)]
```

If an RDD does not have elements for a given key that is present in the other RDD, the corresponding *iterable* is empty. [Listing 4.39](#) shows a `cogroup()` transformation using the `salespeople` and `stores` RDDs from the preceding examples.

Listing 4.39 The `cogroup()` Transformation

[Click here to view code image](#)

```

salespeople.keyBy(lambda x: x[2]) \
    .cogroup(stores).take(1)
# returns:
# [(None, (<pyspark.resultiterable.ResultIterable object at ...>,
# <pyspark.resultiterable.ResultIterable object at ...>))]
salespeople.keyBy(lambda x: x[2]) \
    .cogroup(stores) \
    .mapValues(lambda x: [item for sublist in x for item in
sublist]) \
    .collect()
# using the mapValues() to process the Iterable object returns:
# [(None, [(5, 'Janice', None)]),
# (100, [(1, 'Henry', 100), (2, 'Karen', 100), 'Boca Raton']),
# (102, [(4, 'Jimmy', 102), 'Cambridge']), (101, [(3, 'Paul', 101),
'Columbia']),
# (103, ['Naperville'])]

```

cartesian()

Syntax:

```
RDD.cartesian(<otherRDD>)
```

The `cartesian()` transformation, sometimes referred to by its colloquial name, *cross join*, generates every possible combination of records from both RDDs. The number of records produced by this transformation is equal to the number of records in the first RDD multiplied by the number of records in the second RDD.

[Listing 4.40](#) demonstrates the use of the `cartesian()` transformation.

Listing 4.40 The `cartesian()` Transformation

[Click here to view code image](#)

```

salespeople.keyBy(lambda x: x[2]) \
    .cartesian(stores).take(1)
# returns:
# [((100, (1, 'Henry', 100)), (100, 'Boca Raton'))]
salespeople.keyBy(lambda x: x[2]) \
    .cartesian(stores).count()

```

returns 20 as there are $5 \times 4 = 20$ records

Use the **cartesian()** Transformation Cautiously

Cartesian, or cross-product, operations can yield excessively large amounts of data. Although this is a useful function for testing multiple combinations of items for machine learning, you could create a Big Data problem where one otherwise did not exist!

Joining Datasets in Spark

For this example you will use data from the Bay Area Bike Share Data Challenge. The Bay Area Bike Share program enables members to pick up bikes from designated stations and then drop off the bikes at the same station or a different one. Bay Area Bike Share has made trip data available for public use through the group's Open Data program. For more information, see these sites:

<http://www.bayareabikeshare.com/open-data>

<https://www.fordgobike.com/system-data>

To make your job easier, the data files for this exercise are available in this book's AWS S3 bucket:

<https://s3.amazonaws.com/sparkusingpython/bike-share/stations/stations.csv>

<https://s3.amazonaws.com/sparkusingpython/bike-share/status/status.csv>

<https://s3.amazonaws.com/sparkusingpython/bike-share/trips/trips.csv>

<https://s3.amazonaws.com/sparkusingpython/bike-share/weather/weather.csv>

You can download these files to your local Spark installation and access them locally. For this exercise, you should download the files and store them in your `$SPARK_HOME/data` directory as follows:

```

├── bike-share
│   ├── stations
│   │   └── stations.csv
│   ├── status
│   │   └── status.csv
│   ├── trips
│   │   └── trips.csv
│   └── weather
│       └── weather.csv
└── ...

```

In this exercise, you will use this data to return the average number of bikes available by the hour for one week (February 22 to February 28) for stations located in the San Jose area only. Follow these steps:

1. Open an interactive `pyspark` session:

[Click here to view code image](#)

```
$ pyspark --master local
```

2. Create an RDD named `stations`:

[Click here to view code image](#)

```
stations = sc.textFile('/opt/spark/data/bike-share/stations')
```

[Table 4.2](#) shows the schema or structure of the files in the `stations` directory.

Table 4.2 **Fields in `stations.csv`**

Field Name	Description
<code>station_id</code>	Station ID number
<code>name</code>	Name of the station
<code>lat</code>	Latitude
<code>long</code>	Longitude
<code>dockcount</code>	Number of docks at the station
<code>landmark</code>	City
<code>installation</code>	Original date the station was installed

3. Create an RDD named `status`:

[Click here to view code image](#)

```
status = sc.textFile('/opt/spark/data/bike-share/status')
```

[Table 4.3](#) shows the schema or structure of the files in the `status` directory.

Table 4.3 **Fields in `status.csv`**

Field Name	Description
<code>station_id</code>	Station ID number
<code>bikes_available</code>	Number of available bikes
<code>docks_available</code>	Number of available docks
<code>time</code>	Date and time, PST

4. Split the `status` data into discrete fields, projecting only the fields necessary, and decompose the date string so that you can filter records by date more easily in the next step:

[Click here to view code image](#)

```
status2 = status.map(lambda x: x.split(',')) \
    .map(lambda x: (x[0], x[1], x[2], x[3].replace('"', ''))) \
    .map(lambda x: (x[0], x[1], x[2], x[3].split(' '))) \
    .map(lambda x: (x[0], x[1], x[2], x[3][0].split('-'), x[3]
    [1].split(':'))) \
    .map(lambda x: (int(x[0]), int(x[1]), int(x[3][0]), int(x[3][1]),
    int(x[3][2]), int(x[4][0])))
```

Inspect the `status2` RDD:

```
status2.first()
```

The schema for the `status2` RDD is as follows:

[Click here to view code image](#)

```
[(station_id, bikes_available, year, month, day, hour),...]
```

5. Because `status.csv` is the biggest of the datasets (more than 36 million records), restrict the dataset to only the dates required and then drop the date fields because they are no longer necessary:

[Click here to view code image](#)

```
status3 = status2.filter(lambda x: x[2]==2015 and \
                             x[3]==2 and \
                             x[4]>=22) \
                  .map(lambda x: (x[0], x[1], x[5]))
```

The schema for `status3` is the same as the schema for `status2` because you have just removed unnecessary records.

6. Filter the `stations` dataset to include only stations where `landmark='San Jose'`:

[Click here to view code image](#)

```
stations2 = stations.map(lambda x: x.split(',')) \
                  .filter(lambda x: x[5] == 'San Jose') \
                  .map(lambda x: (int(x[0]), x[1]))
```

Inspect the `stations2` RDD:

```
stations2.first()
```

7. Convert both RDDs to key/value pair RDDs to prepare for a `join()` operation:

[Click here to view code image](#)

```
status_kv = status3.keyBy(lambda x: x[0])
stations_kv = stations2.keyBy(lambda x: x[0])
```

Inspect both newly created PairRDDs:

[Click here to view code image](#)

```
status_kv.first()
stations_kv.first()
```

8. Join the `status_kv` key/value pair RDD to the `stations_kv` key/value pair RDD by their keys (`station_id`):

[Click here to view code image](#)

```
joined = status_kv.join(stations_kv)
```

Inspect the `joined` RDD:

```
joined.first()
```

9. Clean the `joined` RDD:

[Click here to view code image](#)

```
cleaned = joined.map(lambda x: (x[0], x[1][0][1], x[1][0][2], x[1][1][1]))
```

Inspect the `cleaned` RDD:

```
cleaned.first()
```

The schema for the `cleaned` RDD is as follows:

[Click here to view code image](#)

```
[(station_id,bikes_available,hour,name),...]
```

10. Create a key/value pair with the key as a tuple consisting of the station name and the hour and then compute the averages by each hour for each station:

[Click here to view code image](#)

```
avgbyhour = cleaned.keyBy(lambda x: (x[3],x[2])) \
    .mapValues(lambda x: (x[1], 1)) \
    .reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1])) \
    .mapValues(lambda x: (x[0]/x[1]))
```

Inspect the `avgbyhour` RDD:

```
avgbyhour.first()
```

The schema for the `cleaned` RDD is as follows:

[Click here to view code image](#)

```
[((name,hour),bikes_available),...]
```

11. Find the top 10 averages by station and hour by using the `sortBy()` function:

[Click here to view code image](#)

```
topavail = avgbyhour.keyBy(lambda x: x[1]) \
    .sortByKey(ascending=False) \
    .map(lambda x: (x[1][0][0], x[1][0][1], x[0]))
topavail.take(10)
```

The complete source code for this exercise can be found in the `joining-datasets` folder at https://github.com/sparktraining/spark_using_python.

Transformations on Sets

Set operations are conceptually similar to mathematical set operations. A set function operates against two RDDs and results in one RDD. Consider the Venn diagram shown in [Figure 4.9](#), which shows a set of odd integers and a subset of Fibonacci numbers. The following sections use these two sets to demonstrate the various set transformations available in the Spark API.

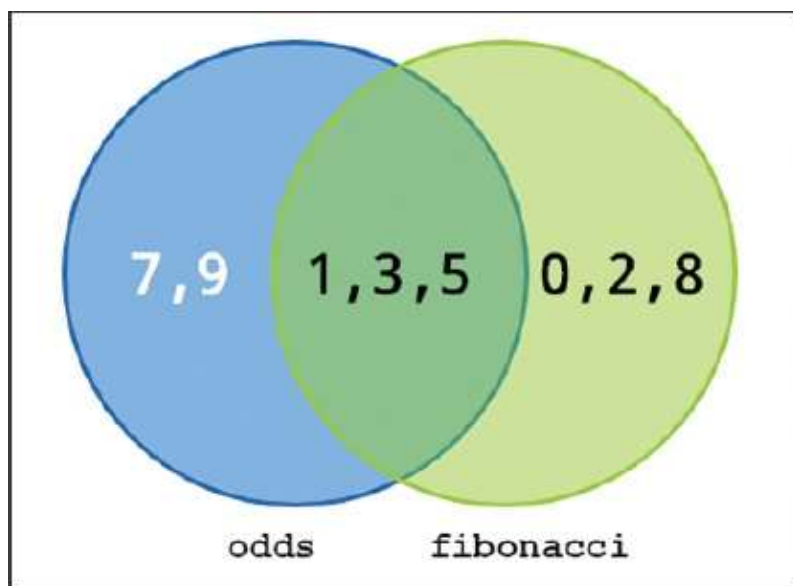


Figure 4.9 Set Venn diagram.

union()

Syntax:

```
RDD.union(<otherRDD>)
```

The `union()` transformation takes one RDD and appends another RDD to it, resulting in a combined output RDD. The RDDs are not required to have the same schema or structure. For instance, the first RDD can have five fields, whereas the second can have more or fewer than five fields.

The `union()` transformation does not filter duplicates from the output RDD in the case that two unioned RDDs have records that are identical to each other. To filter duplicates, you could follow the `union()` transformation with the `distinct()` function discussed previously.

The RDD that results from a `union()` operation is not sorted either, but you could sort it by following `union()` with a `sortBy()` function.

[Listing 4.41](#) shows an example using `union()`.

Listing 4.41 The `union()` Transformation

[Click here to view code image](#)

```
odds = sc.parallelize([1,3,5,7,9])
```



```
fibonacci = sc.parallelize([0,1,2,3,5,8])
odds.union(fibonacci).collect()
# returns [1, 3, 5, 7, 9, 0, 1, 2, 3, 5, 8]
```

intersection()

Syntax:

```
RDD.intersection(<otherRDD>)
```

The `intersection()` transformation returns elements that are present in both RDDs. In other words, it returns the overlap between two sets. The elements or records must be identical in both sets, with each respective record's data structure and all of its fields matching in both RDDs.

[Listing 4.42](#) demonstrates the `intersection()` transformation.

Listing 4.42 The `intersection()` Transformation

[Click here to view code image](#)

```
odds = sc.parallelize([1,3,5,7,9])
fibonacci = sc.parallelize([0,1,2,3,5,8])
odds.intersection(fibonacci).collect()
# returns [1, 3, 5]
```

subtract()

Syntax:

[Click here to view code image](#)

```
RDD.subtract(<otherRDD>, numPartitions=None)
```

The `subtract()` transformation, as shown in [Listing 4.43](#), returns all elements from the first RDD that are not present in the second RDD. This is an implementation of a mathematical set subtraction.

Listing 4.43 The `subtract()` Transformation

[Click here to view code image](#)

```
odds = sc.parallelize([1,3,5,7,9])
```

```
fibonacci = sc.parallelize([0,1,2,3,5,8])
odds.subtract(fibonacci).collect()
# returns [7, 9]
```

subtractByKey()

Syntax:

[Click here to view code image](#)

```
RDD.subtractByKey(<otherRDD>, numPartitions=None)
```

The `subtractByKey()` transformation is a set operation similar to the `subtract` transformation. The `subtractByKey()` transformation returns key/value pair elements from an RDD with keys that are not present in key/value pair elements from `otherRDD`.

The `numPartitions` argument specifies how many output partitions are to be created in the resultant RDD, and it defaults to the configured `spark.default.parallelism` value.

[Listing 4.44](#) demonstrates `subtractByKey()` by using two RDDs containing city names as the key and a tuple containing location data for the city.

Listing 4.44 The `subtractByKey()` Transformation

[Click here to view code image](#)

```
cities1 = sc.parallelize([('Hayward', (37.668819, -122.080795)),
                          ('Baumholder', (49.6489, 7.3975)),
                          ('Alexandria', (38.820450, -77.050552)),
                          ('Melbourne', (37.663712, 144.844788))])
cities2 = sc.parallelize([('Boulder Creek', (64.0708333, -148.2236111)),
                          ('Hayward', (37.668819, -122.080795)),
                          ('Alexandria', (38.820450, -77.050552)),
                          ('Arlington', (38.878337, -77.100703))])
cities1.subtractByKey(cities2).collect()
# returns:
# [('Baumholder', (49.6489, 7.3975)), ('Melbourne', (37.663712,
144.844788))]
cities2.subtractByKey(cities1).collect()
# returns:
```

```
# [('Boulder Creek', (64.0708333, -148.2236111)),  
#  ('Arlington', (38.878337, -77.100703))]
```

Transformations on Numeric RDDs

Numeric RDDs consist of only numeric values. They are commonly used for statistical analysis, so you will see that many of the functions available to numeric RDDs are your common statistical functions. An example of a numeric RDD is the DoubleRDD discussed earlier in this chapter. The following sections look at these functions and provide some simple examples.

min()

Syntax:

```
RDD.min(key=None)
```

The `min()` function is an action that returns the minimum value for a numeric RDD. The `key` argument is a function used to generate a key for comparing.

[Listing 4.45](#) shows the use of the `min()` function.

Listing 4.45 The `min()` Function

[Click here to view code image](#)

```
numbers = sc.parallelize([0,1,1,2,3,5,8,13,21,34])  
numbers.min()  
# returns 0
```

max()

Syntax:

```
RDD.max(key=None)
```

The `max()` function is an action that returns the maximum value for a numeric RDD. The `key` argument is a function used to generate a key for comparing.

[Listing 4.46](#) shows the use of the `max()` function.

Listing 4.46 The `max()` Function

[Click here to view code image](#)

```
numbers = sc.parallelize([0,1,1,2,3,5,8,13,21,34])
numbers.max()
# returns 34
```

`mean()`

Syntax:

`RDD.mean()`

The `mean()` function computes the arithmetic mean from a numeric RDD.

[Listing 4.47](#) demonstrates the use of the `mean()` function.

Listing 4.47 The `mean()` Function

[Click here to view code image](#)

```
numbers = sc.parallelize([0,1,1,2,3,5,8,13,21,34])
numbers.mean()
# returns 8.8
```

`sum()`

Syntax:

`RDD.sum()`

The `sum()` function returns the sum of a list of numbers from a numeric RDD.

[Listing 4.48](#) shows the use of the `sum()` function.

Listing 4.48 The `sum()` Function

[Click here to view code image](#)

```
numbers = sc.parallelize([0,1,1,2,3,5,8,13,21,34])
numbers.sum()
# returns 88
```

stdev()

Syntax:

```
RDD.stdev()
```

The `stdev()` function is an action that computes the standard deviation for a series of numbers from a numeric RDD. [Listing 4.49](#) shows an example of `stdev()`.

Listing 4.49 The `stdev()` Function

[Click here to view code image](#)

```
numbers = sc.parallelize([0,1,1,2,3,5,8,13,21,34])
numbers.stdev()
# returns 10.467091286503619
```

variance()

Syntax:

```
RDD.variance()
```

The `variance()` function computes the variance in a series of numbers in a numeric RDD. Variance is a measure of how far a set of numbers are spread out. [Listing 4.50](#) shows an example of `variance()`.

Listing 4.50 The `variance()` Function

[Click here to view code image](#)

```
numbers = sc.parallelize([0,1,1,2,3,5,8,13,21,34])
numbers.variance()
# returns 109.55999999999999
```

stats()

Syntax:

```
RDD.stats()
```

The `stats()` function returns a `StatCounter` object, which is a structure

containing the `count()`, `mean()`, `stdev()`, `max()`, and `min()` in one operation. [Listing 4.51](#) demonstrates the `stats()` function.

Listing 4.51 The `stats()` Function

[Click here to view code image](#)

```
numbers = sc.parallelize([0,1,1,2,3,5,8,13,21,34])
numbers.stats()
# returns (count: 10, mean: 8.8, stdev: 10.4670912865, max: 34.0, min: 0.0)
```

Summary

This chapter covers the fundamentals of Spark programming, starting with a closer look at Spark RDDs (the most fundamental atomic data object in the Spark programming model), including looking at how to load data into RDDs, how RDDs are evaluated and processed, and how RDDs achieve fault tolerance and resiliency. This chapter also discusses the concepts of transformations and actions in Spark and provides specific descriptions and examples of the most important functions in the Spark core (or RDD) API. This chapter is arguably the most important chapter in this book as it has laid the foundations for all programming in Spark, including stream processing, machine learning, and SQL. The remainder of the book regularly refers to the functions and concepts covered in this chapter.