Machine Learning with Spark

Create scalable machine learning applications to power a modern data-driven business using Spark

Nick Pentreath
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 "Getting Up and Running with Spark"
- A synopsis of the book’s content
- More information on Machine Learning with Spark

About the Author

Nick Pentreath has a background in financial markets, machine learning, and software development. He has worked at Goldman Sachs Group, Inc.; as a research scientist at the online ad targeting start-up Cognitive Match Limited, London; and led the Data Science and Analytics team at Mxit, Africa's largest social network.

He is a cofounder of Graphflow, a big data and machine learning company focused on user-centric recommendations and customer intelligence. He is passionate about combining commercial focus with machine learning and cutting-edge technology to build intelligent systems that learn from data to add value to the bottom line.

Nick is a member of the Apache Spark Project Management Committee.
Acknowledgments

Writing this book has been quite a rollercoaster ride over the past year, with many ups and downs, late nights, and working weekends. It has also been extremely rewarding to combine my passion for machine learning with my love of the Apache Spark project, and I hope to bring some of this out in this book.

I would like to thank the Packt Publishing team for all their assistance throughout the writing and editing process: Rebecca, Susmita, Sudhir, Amey, Neil, Vivek, Pankaj, and everyone who worked on the book.

Thanks also go to Debora Donato at StumbleUpon for assistance with data- and legal-related queries.

Writing a book like this can be a somewhat lonely process, so it is incredibly helpful to get the feedback of reviewers to understand whether one is headed in the right direction (and what course adjustments need to be made). I'm deeply grateful to Andrea Mostosi, Hao Ren, and Krishna Sankar for taking the time to provide such detailed and critical feedback.

I could not have gotten through this project without the unwavering support of all my family and friends, especially my wonderful wife, Tammy, who will be glad to have me back in the evenings and on weekends once again. Thank you all!

Finally, thanks to all of you reading this; I hope you find it useful!
Machine Learning with Spark

In recent years, the volume of data being collected, stored, and analyzed has exploded, in particular in relation to the activity on the Web and mobile devices, as well as data from the physical world collected via sensor networks. While previously large-scale data storage, processing, analysis, and modeling was the domain of the largest institutions such as Google, Yahoo!, Facebook, and Twitter, increasingly, many organizations are being faced with the challenge of how to handle a massive amount of data.

When faced with this quantity of data and the common requirement to utilize it in real time, human-powered systems quickly become infeasible. This has led to a rise in the so-called big data and machine learning systems that learn from this data to make automated decisions.

In answer to the challenge of dealing with ever larger-scale data without any prohibitive cost, new open source technologies emerged at companies such as Google, Yahoo!, Amazon, and Facebook, which aimed at making it easier to handle massive data volumes by distributing data storage and computation across a cluster of computers.

The most widespread of these is Apache Hadoop, which made it significantly easier and cheaper to both store large amounts of data (via the Hadoop Distributed File System, or HDFS) and run computations on this data (via Hadoop MapReduce, a framework to perform computation tasks in parallel across many nodes in a computer cluster).

However, MapReduce has some important shortcomings, including high overheads to launch each job and reliance on storing intermediate data and results of the computation to disk, both of which make Hadoop relatively ill-suited for use cases of an iterative or low-latency nature. Apache Spark is a new framework for distributed computing that is designed from the ground up to be optimized for low-latency tasks and to store intermediate data and results in memory, thus addressing some of the major drawbacks of the Hadoop framework. Spark provides a clean, functional, and easy-to-understand API to write applications and is fully compatible with the Hadoop ecosystem.

Furthermore, Spark provides native APIs in Scala, Java, and Python. The Scala and Python APIs allow all the benefits of the Scala or Python language, respectively, to be used directly in Spark applications, including using the relevant interpreter for real-time, interactive exploration. Spark itself now provides a toolkit (called MLlib) of distributed machine learning and data mining models that is under heavy development and already contains high-quality, scalable, and efficient algorithms for many common machine learning tasks, some of which we will delve into in this book.
Applying machine learning techniques to massive datasets is challenging, primarily because most well-known machine learning algorithms are not designed for parallel architectures. In many cases, designing such algorithms is not an easy task. The nature of machine learning models is generally iterative, hence the strong appeal of Spark for this use case. While there are many competing frameworks for parallel computing, Spark is one of the few that combines speed, scalability, in-memory processing, and fault tolerance with ease of programming and a flexible, expressive, and powerful API design.

Throughout this book, we will focus on real-world applications of machine learning technology. While we may briefly delve into some theoretical aspects of machine learning algorithms, the book will generally take a practical, applied approach with a focus on using examples and code to illustrate how to effectively use the features of Spark and MLlib, as well as other well-known and freely available packages for machine learning and data analysis, to create a useful machine learning system.

**What This Book Covers**

*Chapter 1, Getting Up and Running with Spark*, shows how to install and set up a local development environment for the Spark framework as well as how to create a Spark cluster in the cloud using Amazon EC2. The Spark programming model and API will be introduced, and a simple Spark application will be created using each of Scala, Java, and Python.

*Chapter 2, Designing a Machine Learning System*, presents an example of a real-world use case for a machine learning system. We will design a high-level architecture for an intelligent system in Spark based on this illustrative use case.

*Chapter 3, Obtaining, Processing, and Preparing Data with Spark*, details how to go about obtaining data for use in a machine learning system, in particular from various freely and publicly available sources. We will learn how to process, clean, and transform the raw data into features that may be used in machine learning models, using available tools, libraries, and Spark's functionality.

*Chapter 4, Building a Recommendation Engine with Spark*, deals with creating a recommendation model based on the collaborative filtering approach. This model will be used to recommend items to a given user as well as create lists of items that are similar to a given item. Standard metrics to evaluate the performance of a recommendation model will be covered here.

*Chapter 5, Building a Classification Model with Spark*, details how to create a model for binary classification as well as how to utilize standard performance-evaluation metrics for classification tasks.
Chapter 6, *Building a Regression Model with Spark*, shows how to create a model for regression, extending the classification model created in *Chapter 5, Building a Classification Model with Spark*. Evaluation metrics for the performance of regression models will be detailed here.

Chapter 7, *Building a Clustering Model with Spark*, explores how to create a clustering model as well as how to use related evaluation methodologies. You will learn how to analyze and visualize the clusters generated.

Chapter 8, *Dimensionality Reduction with Spark*, takes us through methods to extract the underlying structure from and reduce the dimensionality of our data. You will learn some common dimensionality-reduction techniques and how to apply and analyze them, as well as how to use the resulting data representation as input to another machine learning model.

Chapter 9, *Advanced Text Processing with Spark*, introduces approaches to deal with large-scale text data, including techniques for feature extraction from text and dealing with the very high-dimensional features typical in text data.

Chapter 10, *Real-time Machine Learning with Spark Streaming*, provides an overview of Spark Streaming and how it fits in with the online and incremental learning approaches to apply machine learning on data streams.
Getting Up and Running with Spark

Apache Spark is a framework for distributed computing; this framework aims to make it simpler to write programs that run in parallel across many nodes in a cluster of computers. It tries to abstract the tasks of resource scheduling, job submission, execution, tracking, and communication between nodes, as well as the low-level operations that are inherent in parallel data processing. It also provides a higher level API to work with distributed data. In this way, it is similar to other distributed processing frameworks such as Apache Hadoop; however, the underlying architecture is somewhat different.

Spark began as a research project at the University of California, Berkeley. The university was focused on the use case of distributed machine learning algorithms. Hence, it is designed from the ground up for high performance in applications of an iterative nature, where the same data is accessed multiple times. This performance is achieved primarily through caching datasets in memory, combined with low latency and overhead to launch parallel computation tasks. Together with other features such as fault tolerance, flexible distributed-memory data structures, and a powerful functional API, Spark has proved to be broadly useful for a wide range of large-scale data processing tasks, over and above machine learning and iterative analytics.

For more background on Spark, including the research papers underlying Spark’s development, see the project's history page at [http://spark.apache.org/community.html#history](http://spark.apache.org/community.html#history).
Spark runs in four modes:

- The standalone local mode, where all Spark processes are run within the same Java Virtual Machine (JVM) process
- The standalone cluster mode, using Spark's own built-in job-scheduling framework
- Using Mesos, a popular open source cluster-computing framework
- Using YARN (commonly referred to as NextGen MapReduce), a Hadoop-related cluster-computing and resource-scheduling framework

In this chapter, we will:

- Download the Spark binaries and set up a development environment that runs in Spark's standalone local mode. This environment will be used throughout the rest of the book to run the example code.
- Explore Spark's programming model and API using Spark's interactive console.
- Write our first Spark program in Scala, Java, and Python.
- Set up a Spark cluster using Amazon's Elastic Cloud Compute (EC2) platform, which can be used for large-sized data and heavier computational requirements, rather than running in the local mode.

Spark can also be run on Amazon's Elastic MapReduce service using custom bootstrap action scripts, but this is beyond the scope of this book. The following article is a good reference guide: [http://aws.amazon.com/articles/Elastic-MapReduce/4926593393724923](http://aws.amazon.com/articles/Elastic-MapReduce/4926593393724923). At the time of writing this book, the article covers running Spark Version 1.1.0.

If you have previous experience in setting up Spark and are familiar with the basics of writing a Spark program, feel free to skip this chapter.

### Installing and setting up Spark locally

Spark can be run using the built-in standalone cluster scheduler in the local mode. This means that all the Spark processes are run within the same JVM—effectively, a single, multithreaded instance of Spark. The local mode is very useful for prototyping, development, debugging, and testing. However, this mode can also be useful in real-world scenarios to perform parallel computation across multiple cores on a single computer.
As Spark's local mode is fully compatible with the cluster mode, programs written and tested locally can be run on a cluster with just a few additional steps.

The first step in setting up Spark locally is to download the latest version (at the time of writing this book, the version is 1.2.0). The download page of the Spark project website, found at http://spark.apache.org/downloads.html, contains links to download various versions as well as to obtain the latest source code via GitHub.

The Spark project documentation website at http://spark.apache.org/docs/latest/ is a comprehensive resource to learn more about Spark. We highly recommend that you explore it!

Spark needs to be built against a specific version of Hadoop in order to access Hadoop Distributed File System (HDFS) as well as standard and custom Hadoop input sources. The download page provides prebuilt binary packages for Hadoop 1, CDH4 (Cloudera's Hadoop Distribution), MapR's Hadoop distribution, and Hadoop 2 (YARN). Unless you wish to build Spark against a specific Hadoop version, we recommend that you download the prebuilt Hadoop 2.4 package from an Apache mirror using this link: http://www.apache.org/dyn/closer.cgi/spark/spark-1.2.0/spark-1.2.0-bin-hadoop2.4.tgz.

Spark requires the Scala programming language (version 2.10.4 at the time of writing this book) in order to run. Fortunately, the prebuilt binary package comes with the Scala runtime packages included, so you don't need to install Scala separately in order to get started. However, you will need to have a Java Runtime Environment (JRE) or Java Development Kit (JDK) installed (see the software and hardware list in this book's code bundle for installation instructions).

Once you have downloaded the Spark binary package, unpack the contents of the package and change into the newly created directory by running the following commands:

```
>tar xfvz spark-1.2.0-bin-hadoop2.4.tgz
>cd spark-1.2.0-bin-hadoop2.4
```

Spark places user scripts to run Spark in the bin directory. You can test whether everything is working correctly by running one of the example programs included in Spark:

```
>./bin/run-example org.apache.spark.examples.SparkPi
```
This will run the example in Spark's local standalone mode. In this mode, all the Spark processes are run within the same JVM, and Spark uses multiple threads for parallel processing. By default, the preceding example uses a number of threads equal to the number of cores available on your system. Once the program is finished running, you should see something similar to the following lines near the end of the output:

```
14/11/27 20:58:47 INFO SparkContext: Job finished: reduce at SparkPi.
scala:35, took 0.723269 s
Pi is roughly 3.1465
```

To configure the level of parallelism in the local mode, you can pass in a master parameter of the `local[N]` form, where N is the number of threads to use. For example, to use only two threads, run the following command instead:

```
>MASTER=local[2] ./bin/run-example org.apache.spark.examples.SparkPi
```

### Spark clusters

A Spark cluster is made up of two types of processes: a driver program and multiple executors. In the local mode, all these processes are run within the same JVM. In a cluster, these processes are usually run on separate nodes.

For example, a typical cluster that runs in Spark's standalone mode (that is, using Spark's built-in cluster-management modules) will have:

- A master node that runs the Spark standalone master process as well as the driver program
- A number of worker nodes, each running an executor process

While we will be using Spark's local standalone mode throughout this book to illustrate concepts and examples, the same Spark code that we write can be run on a Spark cluster. In the preceding example, if we run the code on a Spark standalone cluster, we could simply pass in the URL for the master node as follows:

```
>MASTER=spark://IP:PORT ./bin/run-example org.apache.spark.examples.SparkPi
```
Here, IP is the IP address, and PORT is the port of the Spark master. This tells Spark to run the program on the cluster where the Spark master process is running.

A full treatment of Spark's cluster management and deployment is beyond the scope of this book. However, we will briefly teach you how to set up and use an Amazon EC2 cluster later in this chapter.

For an overview of the Spark cluster-application deployment, take a look at the following links:

- [http://spark.apache.org/docs/latest/cluster-overview.html](http://spark.apache.org/docs/latest/cluster-overview.html)
- [http://spark.apache.org/docs/latest/submitting-applications.html](http://spark.apache.org/docs/latest/submitting-applications.html)

### The Spark programming model

Before we delve into a high-level overview of Spark's design, we will introduce the SparkContext object as well as the Spark shell, which we will use to interactively explore the basics of the Spark programming model.

While this section provides a brief overview and examples of using Spark, we recommend that you read the following documentation to get a detailed understanding:

- Spark Quick Start: [http://spark.apache.org/docs/latest/quick-start.html](http://spark.apache.org/docs/latest/quick-start.html)

### SparkContext and SparkConf

The starting point of writing any Spark program is SparkContext (or JavaSparkContext in Java). SparkContext is initialized with an instance of a SparkConf object, which contains various Spark cluster-configuration settings (for example, the URL of the master node).
Once initialized, we will use the various methods found in the SparkContext object to create and manipulate distributed datasets and shared variables. The Spark shell (in both Scala and Python, which is unfortunately not supported in Java) takes care of this context initialization for us, but the following lines of code show an example of creating a context running in the local mode in Scala:

```scala
val conf = new SparkConf()
  .setAppName("Test Spark App")
  .setMaster("local[4]")
val sc = new SparkContext(conf)
```

This creates a context running in the local mode with four threads, with the name of the application set to Test Spark App. If we wish to use default configuration values, we could also call the following simple constructor for our SparkContext object, which works in exactly the same way:

```scala
val sc = new SparkContext("local[4]", "Test Spark App")
```

**The Spark shell**

Spark supports writing programs interactively using either the Scala or Python REPL (that is, the Read-Eval-Print-Loop, or interactive shell). The shell provides instant feedback as we enter code, as this code is immediately evaluated. In the Scala shell, the return result and type is also displayed after a piece of code is run.
To use the Spark shell with Scala, simply run `.bin/spark-shell` from the Spark base directory. This will launch the Scala shell and initialize `SparkContext`, which is available to us as the Scala value, `sc`. Your console output should look similar to the following screenshot:

```
Nicks-MacBook-Pro:spark-1.2.0-bin-hadoop2.4 Nick$ .bin/spark-shell
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
14/11/27 22:02:26 INFO SecurityManager: Changing view acls to: Nick
14/11/27 22:02:26 INFO SecurityManager: Changing modify acls to: Nick
14/11/27 22:02:26 INFO SecurityManager: SecurityManager: authentication disabled; ui acls disabled; users with view permissions: Set(Nick); users with modify permissions: Set(Nick)
14/11/27 22:02:26 INFO SparkServer: Starting HTTP Server
14/11/27 22:02:26 INFO Utils: Successfully started service 'HTTP class server' on port 55290.
Welcome to

  / scala 1.3.0 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_68)

Using Scala version 2.10.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_68)
Type in expressions to have them evaluated.
Type :help for more information.
14/11/27 22:02:39 WARN Util: Your hostname, Nicks-MacBook-Pro, local resolves to a loopback address: 127.0.0.1; using 127.0.0.1 instead (on interface en0)
14/11/27 22:02:39 WARN Util: Set SPARK_LOCAL_IP if you need to bind to another address
14/11/27 22:02:39 INFO SecurityManager: Changing view acls to: Nick
14/11/27 22:02:39 INFO SecurityManager: Changing modify acls to: Nick
14/11/27 22:02:39 INFO SecurityManager: SecurityManager: authentication disabled; ui acls disabled; users with view permissions: Set(Nick); users with modify permissions: Set(Nick)
14/11/27 22:02:39 INFO S14Logger: S14Logger started
14/11/27 22:02:39 INFO Remote: Starting remote
14/11/27 22:02:39 INFO Remote: Remote started; listening on addresses :[akka.tcp://sparkDriver@10.0.0.7:55290]
14/11/27 22:02:39 INFO Util: Successfully started service 'sparkDriver' on port 55290.
14/11/27 22:02:39 INFO SparkEnv: Registering BlockManagerMaster
14/11/27 22:02:39 INFO DiskBlockManager: Created local directory at /var/folders/_v/06w1jt3wqg7r0j1c44_r0000gn/T/spark-local-20141127220232-634b
14/11/27 22:02:39 INFO MemoryStore: MemoryStore started with capacity 265.4 MB
14/11/27 22:02:39 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using built-in-java classes where applicable
14/11/27 22:02:39 INFO FileServer: HTTP file server directory is /var/folders/_v/06w1jt3wqg7r0j1c44_r0000gn/T/spark-10950159-f23f-40bd-8ca5-507d65543e55
14/11/27 22:02:39 INFO FileServer: Starting HTTP Server
14/11/27 22:02:32 INFO Executor: Using REPL class URI: http://10.0.0.7:55288
14/11/27 22:02:32 INFO AkkaUtils: Connecting to HeartbeatReceiver: akka.tcp://sparkDriver@10.0.0.7:55290/user/HeartbeatReceiver
14/11/27 22:02:32 INFO NettyBlockTransferService: Server created on 55292
14/11/27 22:02:32 INFO BlockManagerMaster: Trying to register BlockManager
14/11/27 22:02:32 INFO BlockManagerMaster: Registering block manager localhost:55292 with 265.4 MB RAM, BlockManager Executor ID=start, localhost, 55292)
14/11/27 22:02:32 INFO BlockManagerMaster: Registered BlockManager
14/11/27 22:02:32 INFO SparkLoop: Created spark context.

Spark context available as sc.
```
To use the Python shell with Spark, simply run the `./bin/pyspark` command. Like the Scala shell, the Python SparkContext object should be available as the Python variable `sc`. You should see an output similar to the one shown in this screenshot:

```
Resilient Distributed Datasets

The core of Spark is a concept called the Resilient Distributed Dataset (RDD). An RDD is a collection of "records" (strictly speaking, objects of some type) that is distributed or partitioned across many nodes in a cluster (for the purposes of the Spark local mode, the single multithreaded process can be thought of in the same way). An RDD in Spark is fault-tolerant; this means that if a given node or task fails (for some reason other than erroneous user code, such as hardware failure, loss of communication, and so on), the RDD can be reconstructed automatically on the remaining nodes and the job will still complete.

```

Resilient Distributed Datasets

The core of Spark is a concept called the Resilient Distributed Dataset (RDD). An RDD is a collection of "records" (strictly speaking, objects of some type) that is distributed or partitioned across many nodes in a cluster (for the purposes of the Spark local mode, the single multithreaded process can be thought of in the same way). An RDD in Spark is fault-tolerant; this means that if a given node or task fails (for some reason other than erroneous user code, such as hardware failure, loss of communication, and so on), the RDD can be reconstructed automatically on the remaining nodes and the job will still complete.
Creating RDDs

RDDs can be created from existing collections, for example, in the Scala Spark shell that you launched earlier:

```scala
val collection = List("a", "b", "c", "d", "e")
val rddFromCollection = sc.parallelize(collection)
```

RDDs can also be created from Hadoop-based input sources, including the local filesystem, HDFS, and Amazon S3. A Hadoop-based RDD can utilize any input format that implements the Hadoop `InputFormat` interface, including text files, other standard Hadoop formats, HBase, Cassandra, and many more. The following code is an example of creating an RDD from a text file located on the local filesystem:

```scala
val rddFromTextFile = sc.textFile("LICENSE")
```

The preceding `textFile` method returns an RDD where each record is a `String` object that represents one line of the text file.

Spark operations

Once we have created an RDD, we have a distributed collection of records that we can manipulate. In Spark's programming model, operations are split into transformations and actions. Generally speaking, a transformation operation applies some function to all the records in the dataset, changing the records in some way. An action typically runs some computation or aggregation operation and returns the result to the driver program where `SparkContext` is running.

Spark operations are functional in style. For programmers familiar with functional programming in Scala or Python, these operations should seem natural. For those without experience in functional programming, don't worry; the Spark API is relatively easy to learn.

One of the most common transformations that you will use in Spark programs is the `map` operator. This applies a function to each record of an RDD, thus mapping the input to some new output. For example, the following code fragment takes the RDD we created from a local text file and applies the `size` function to each record in the RDD. Remember that we created an RDD of `Strings`. Using `map`, we can transform each string to an integer, thus returning an RDD of `Ints`:

```scala
val intsFromStringsRDD = rddFromTextFile.map(line => line.size)
```
You should see output similar to the following line in your shell; this indicates the type of the RDD:

```

In the preceding code, we saw the `=>` syntax used. This is the Scala syntax for an anonymous function, which is a function that is not a named method (that is, one defined using the `def` keyword in Scala or Python, for example).

While a detailed treatment of anonymous functions is beyond the scope of this book, they are used extensively in Spark code in Scala and Python, as well as in Java 8 (both in examples and real-world applications), so it is useful to cover a few practicalities.

The `line => line.size` syntax means that we are applying a function where the input variable is to the left of the `=>` operator, and the output is the result of the code to the right of the `=>` operator. In this case, the input is `line`, and the output is the result of calling `line.size`. In Scala, this function that maps a string to an integer is expressed as `String => Int`. This syntax saves us from having to separately define functions every time we use methods such as `map`; this is useful when the function is simple and will only be used once, as in this example.

Now, we can apply a common action operation, `count`, to return the number of records in our RDD:

```text
intsFromStringsRDD.count
```

The result should look something like the following console output:

```
...
14/01/29 23:28:28 INFO SparkContext: Job finished: count at <console>:17, took 0.019227 s
res4: Long = 398
```

Perhaps we want to find the average length of each line in this text file. We can first use the `sum` function to add up all the lengths of all the records and then divide the sum by the number of records:

```text
val sumOfRecords = intsFromStringsRDD.sum
val numRecords = intsFromStringsRDD.count
val aveLengthOfRecord = sumOfRecords / numRecords
```
The result will be as follows:

```scala
calveLengthOfRecord: Double = 52.06030150753769
```

Spark operations, in most cases, return a new RDD, with the exception of most actions, which return the result of a computation (such as `count` and `sum` for `Long` and `Double` in the preceding example). This means that we can naturally chain together operations to make our program flow more concise and expressive. For example, the same result as the one in the preceding line of code can be achieved using the following code:

```scala
val aveLengthOfRecordChained = rddFromTextFile.map(line => line.size).sum / rddFromTextFile.count
```

An important point to note is that Spark transformations are lazy. That is, invoking a transformation on an RDD does not immediately trigger a computation. Instead, transformations are chained together and are effectively only computed when an action is called. This allows Spark to be more efficient by only returning results to the driver when necessary so that the majority of operations are performed in parallel on the cluster.

This means that if your Spark program never uses an action operation, it will never trigger an actual computation, and you will not get any results. For example, the following code will simply return a new RDD that represents the chain of transformations:

```scala
val transformedRDD = rddFromTextFile.map(line => line.size).filter(size => size > 10).map(size => size * 2)
```

This returns the following result in the console:

```
```

Notice that no actual computation happens and no result is returned. If we now call an action, such as `sum`, on the resulting RDD, the computation will be triggered:

```scala
val computation = transformedRDD.sum
```

You will now see that a Spark job is run, and it results in the following console output:

```
14/11/27 21:48:21 INFO SparkContext: Job finished: sum at <console>:16, took 0.193513 s
computation: Double = 60468.0
```
The complete list of transformations and actions possible on RDDs as well as a set of more detailed examples are available in the Spark programming guide (located at http://spark.apache.org/docs/latest/programming-guide.html#rdd-operations), and the API documentation (the Scala API documentation) is located at http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.rdd.RDD).

Caching RDDs

One of the most powerful features of Spark is the ability to cache data in memory across a cluster. This is achieved through use of the cache method on an RDD:

```scala
rddFromTextFile.cache
```

Calling cache on an RDD tells Spark that the RDD should be kept in memory. The first time an action is called on the RDD that initiates a computation, the data is read from its source and put into memory. Hence, the first time such an operation is called, the time it takes to run the task is partly dependent on the time it takes to read the data from the input source. However, when the data is accessed the next time (for example, in subsequent queries in analytics or iterations in a machine learning model), the data can be read directly from memory, thus avoiding expensive I/O operations and speeding up the computation, in many cases, by a significant factor.

If we now call the count or sum function on our cached RDD, we will see that the RDD is loaded into memory:

```scala
val aveLengthOfRecordChained = rddFromTextFile.map(line => line.size).sum / rddFromTextFile.count
```

Indeed, in the following output, we see that the dataset was cached in memory on the first call, taking up approximately 62 KB and leaving us with around 270 MB of memory free:

```
14/01/30 06:59:27 INFO MemoryStore: ensureFreeSpace(63454) called with curMem=32960, maxMem=311387750
14/01/30 06:59:27 INFO MemoryStore: Block rdd_2_0 stored as values to memory (estimated size 62.0 KB, free 296.9 MB)
14/01/30 06:59:27 INFO BlockManagerMasterActor$BlockManagerInfo: Added rdd_2_0 in memory on 10.0.0.3:55089 (size: 62.0 KB, free: 296.9 MB)
...
Now, we will call the same function again:

```scala
val aveLengthOfRecordChainedFromCached = rddFromTextFile.map(line => line.size).sum / rddFromTextFile.count
```

We will see from the console output that the cached data is read directly from memory:

```
14/01/30 06:59:34 INFO BlockManager: Found block rdd_2_0 locally
```

Spark also allows more fine-grained control over caching behavior. You can use the `persist` method to specify what approach Spark uses to cache data. More information on RDD caching can be found here: http://spark.apache.org/docs/latest/programming-guide.html#rdd-persistence.

### Broadcast variables and accumulators

Another core feature of Spark is the ability to create two special types of variables: broadcast variables and accumulators.

A **broadcast variable** is a read-only variable that is made available from the driver program that runs the `SparkContext` object to the nodes that will execute the computation. This is very useful in applications that need to make the same data available to the worker nodes in an efficient manner, such as machine learning algorithms. Spark makes creating broadcast variables as simple as calling a method on `SparkContext` as follows:

```scala
val broadcastAList = sc.broadcast(List("a", "b", "c", "d", "e"))
```

The console output shows that the broadcast variable was stored in memory, taking up approximately 488 bytes, and it also shows that we still have 270 MB available to us:

```
14/01/30 07:13:32 INFO MemoryStore: ensureFreeSpace(488) called with curMem=96414, maxMem=311387750
14/01/30 07:13:32 INFO MemoryStore: Block broadcast_1 stored as values to memory (estimated size 488.0 B, free 296.9 MB)
broadcastAList: org.apache.spark.broadcast.Broadcast[List[String]] = Broadcast(1)
```
A broadcast variable can be accessed from nodes other than the driver program that created it (that is, the worker nodes) by calling `value` on the variable:

```scala
sc.parallelize(List("1", "2", "3").map(x => broadcastAList.value ++ x)).collect
```

This code creates a new RDD with three records from a collection (in this case, a Scala `List`) of ("1", "2", "3"). In the `map` function, it returns a new collection with the relevant record from our new RDD appended to the `broadcastAList` that is our broadcast variable.

Notice that we used the `collect` method in the preceding code. This is a Spark `action` that returns the entire RDD to the driver as a Scala (or Python or Java) collection.

We will often use `collect` when we wish to apply further processing to our results locally within the driver program.

Note that `collect` should generally only be used in cases where we really want to return the full result set to the driver and perform further processing. If we try to call `collect` on a very large dataset, we might run out of memory on the driver and crash our program.

It is preferable to perform as much heavy-duty processing on our Spark cluster as possible, preventing the driver from becoming a bottleneck. In many cases, however, collecting results to the driver is necessary, such as during iterations in many machine learning models.

On inspecting the result, we will see that for each of the three records in our new RDD, we now have a record that is our original broadcasted `List`, with the new element appended to it (that is, there is now either "1", "2", or "3" at the end):

```
14/01/31 10:15:39 INFO SparkContext: Job finished: collect at <console>:15, took 0.025806 s
res6: Array[List[Array]] = Array(List(a, b, c, d, e, 1), List(a, b, c, d, e, 2), List(a, b, c, d, e, 3))
```

An `accumulator` is also a variable that is broadcasted to the worker nodes. The key difference between a broadcast variable and an accumulator is that while the broadcast variable is read-only, the accumulator can be added to. There are limitations to this, that is, in particular, the addition must be an associative operation so that the global accumulated value can be correctly computed in parallel and returned to the driver program. Each worker node can only access and add to its own local accumulator value, and only the driver program can access the global value. Accumulators are also accessed within the Spark code using the `value` method.
For more details on broadcast variables and accumulators, see the Shared Variables section of the Spark Programming Guide: http://spark.apache.org/docs/latest/programming-guide.html#shared-variables.

The first step to a Spark program in Scala

We will now use the ideas we introduced in the previous section to write a basic Spark program to manipulate a dataset. We will start with Scala and then write the same program in Java and Python. Our program will be based on exploring some data from an online store, about which users have purchased which products. The data is contained in a comma-separated-value (CSV) file called UserPurchaseHistory.csv, and the contents are shown in the following snippet. The first column of the CSV is the username, the second column is the product name, and the final column is the price:

John,iPhone Cover,9.99
John,Headphones,5.49
Jack,iPhone Cover,9.99
Jill,Samsung Galaxy Cover,8.95
Bob,iPad Cover,5.49

For our Scala program, we need to create two files: our Scala code and our project build configuration file, using the build tool Scala Build Tool (sbt). For ease of use, we recommend that you download the sample project code called scala-spark-app for this chapter. This code also contains the CSV file under the data directory. You will need SBT installed on your system in order to run this example program (we use version 0.13.1 at the time of writing this book).

Setting up SBT is beyond the scope of this book; however, you can find more information at http://www.scala-sbt.org/release/docs/Getting-Started/Setup.html.

Our SBT configuration file, build.sbt, looks like this (note that the empty lines between each line of code are required):

```scala
name := "scala-spark-app"

version := "1.0"
```
Getting Up and Running with Spark

scalaVersion := "2.10.4"

libraryDependencies += "org.apache.spark" %% "spark-core" % "1.2.0"

The last line adds the dependency on Spark to our project.

Our Scala program is contained in the ScalaApp.scala file. We will walk through the program piece by piece. First, we need to import the required Spark classes:

```scala
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
```

```scala
/**
 * A simple Spark app in Scala
 */
object ScalaApp {
```

In our main method, we need to initialize our SparkContext object and use this to access our CSV data file with the textFile method. We will then map the raw text by splitting the string on the delimiter character (a comma in this case) and extracting the relevant records for username, product, and price:

```scala
def main(args: Array[String]) {
  val sc = new SparkContext("local[2]", "First Spark App")
  // we take the raw data in CSV format and convert it into a set of records of the form (user, product, price)
  val data = sc.textFile("data/UserPurchaseHistory.csv")
    .map(line => line.split(","))
    .map(purchaseRecord => (purchaseRecord(0), purchaseRecord(1), purchaseRecord(2)))
```

Now that we have an RDD, where each record is made up of (user, product, price), we can compute various interesting metrics for our store, such as the following ones:

- The total number of purchases
- The number of unique users who purchased
- Our total revenue
- Our most popular product
Let's compute the preceding metrics:

```scala
// let's count the number of purchases
val numPurchases = data.count()

// let's count how many unique users made purchases
val uniqueUsers = data.map{ case (user, product, price) => user
}.distinct().count()

// let's sum up our total revenue
val totalRevenue = data.map{ case (user, product, price) => price.
toDouble 
}.sum()

// let's find our most popular product
val productsByPopularity = data
.map{ case (user, product, price) => (product, 1) }
.reduceByKey(_ + _)
.collect()
.sortBy(-_._2)
val mostPopular = productsByPopularity(0)
```

This last piece of code to compute the most popular product is an example of the *Map/Reduce* pattern made popular by Hadoop. First, we mapped our records of `(user, product, price)` to the records of `(product, 1)`. Then, we performed a `reduceByKey` operation, where we summed up the 1s for each unique product.

Once we have this transformed RDD, which contains the number of purchases for each product, we will call `collect`, which returns the results of the computation to the driver program as a local Scala collection. We will then sort these counts locally (note that in practice, if the amount of data is large, we will perform the sorting in parallel, usually with a Spark operation such as `sortByKey`).

Finally, we will print out the results of our computations to the console:

```scala
println("Total purchases: " + numPurchases)
println("Unique users: " + uniqueUsers)
println("Total revenue: " + totalRevenue)
println("Most popular product: %s with %d purchases".
format(mostPopular._1, mostPopular._2))
```

We can run this program by running `sbt run` in the project's base directory or by running the program in your Scala IDE if you are using one. The output should look similar to the following:

```
...  
[info] Compiling 1 Scala source to ...
```
Getting Up and Running with Spark

[info] Running ScalaApp
...
14/01/30 10:54:40 INFO spark.SparkContext: Job finished: collect at ScalaApp.scala:25, took 0.045181 s
Total purchases: 5
Unique users: 4
Total revenue: 39.91
Most popular product: iPhone Cover with 2 purchases

We can see that we have five purchases from four different users with a total revenue of 39.91. Our most popular product is an iPhone cover with 2 purchases.

The first step to a Spark program in Java

The Java API is very similar in principle to the Scala API. However, while Scala can call the Java code quite easily, in some cases, it is not possible to call the Scala code from Java. This is particularly the case when such Scala code makes use of certain Scala features such as implicit conversions, default parameters, and the Scala reflection API.

Spark makes heavy use of these features in general, so it is necessary to have a separate API specifically for Java that includes Java versions of the common classes. Hence, SparkContext becomes JavaSparkContext, and RDD becomes JavaRDD.

Java versions prior to version 8 do not support anonymous functions and do not have succinct syntax for functional-style programming, so functions in the Spark Java API must implement a WrappedFunction interface with the call method signature. While it is significantly more verbose, we will often create one-off anonymous classes to pass to our Spark operations, which implement this interface and the call method, to achieve much the same effect as anonymous functions in Scala.

Spark provides support for Java 8's anonymous function (or lambda) syntax. Using this syntax makes a Spark program written in Java 8 look very close to the equivalent Scala program.

In Scala, an RDD of key/value pairs provides special operators (such as reduceByKey and saveAsSequenceFile, for example) that are accessed automatically via implicit conversions. In Java, special types of JavaRDD classes are required in order to access similar functions. These include JavaPairRDD to work with key/value pairs and JavaDoubleRDD to work with numerical records.
In this section, we covered the standard Java API syntax. For more details and examples related to working RDDs in Java as well as the Java 8 lambda syntax, see the Java sections of the Spark Programming Guide found at http://spark.apache.org/docs/latest/programming-guide.html#rdd-operations.

We will see examples of most of these differences in the following Java program, which is included in the example code of this chapter in the directory named java-spark-app. The code directory also contains the CSV data file under the data subdirectory.

We will build and run this project with the Maven build tool, which we assume you have installed on your system.

Installing and setting up Maven is beyond the scope of this book. Usually, Maven can easily be installed using the package manager on your Linux system or HomeBrew or MacPorts on Mac OS X. Detailed installation instructions can be found here: http://maven.apache.org/download.cgi.

The project contains a Java file called JavaApp.java, which contains our program code:

```java
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.api.java.function.DoubleFunction;
import org.apache.spark.api.java.function.Function;
import org.apache.spark.api.java.function.Function2;
import org.apache.spark.api.java.function.PairFunction;
import scala.Tuple2;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

/**
 * A simple Spark app in Java
 */
public class JavaApp {
    public static void main(String[] args) {
```
As in our Scala example, we first need to initialize our context. Notice that we will use the JavaSparkContext class here instead of the SparkContext class that we used earlier. We will use the JavaSparkContext class in the same way to access our data using `textFile` and then split each row into the required fields. Note how we used an anonymous class to define a split function that performs the string processing, in the highlighted code:

```java
JavaSparkContext sc = new JavaSparkContext("local[2]",
"First Spark App");
// we take the raw data in CSV format and convert it into a
set of records of the form (user, product, price)
JavaRDD<String[]> data =
  sc.textFile("data/UserPurchaseHistory.csv")
  .map(new Function<String, String[]>() {
    @Override
    public String[] call(String s) throws Exception {
      return s.split(",");
    }
  });
```

Now, we can compute the same metrics as we did in our Scala example. Note how some methods are the same (for example, `distinct` and `count`) for the Java and Scala APIs. Also note the use of anonymous classes that we pass to the `map` function. This code is highlighted here:

```java
// let's count the number of purchases
long numPurchases = data.count();
// let's count how many unique users made purchases
long uniqueUsers = data.map(new Function<String[], String>() {
  @Override
  public String call(String[] strings) throws Exception {
    return strings[0];
  }
}).distinct().count();
// let's sum up our total revenue
double totalRevenue = data.map(new DoubleFunction<String[]>() {
  @Override
  public Double call(String[] strings) throws Exception {
    return Double.parseDouble(strings[2]);
  }
}).sum();
```
In the following lines of code, we can see that the approach to compute the most popular product is the same as that in the Scala example. The extra code might seem complex, but it is mostly related to the Java code required to create the anonymous functions (which we have highlighted here). The actual functionality is the same:

```java
// let's find our most popular product
// first we map the data to records of (product, 1)
// using a PairFunction
// and the Tuple2 class.
// then we call a reduceByKey operation with a Function2,
// which is essentially the sum function
List<Tuple2<String, Integer>> pairs = data.map(new PairFunction<String[], String, Integer>() {
    @Override
    public Tuple2<String, Integer> call(String[] strings)
    throws Exception {
        return new Tuple2(strings[1], 1);
    }
}).reduceByKey(new Function2<Integer, Integer, Integer>() {
    @Override
    public Integer call(Integer integer, Integer integer2)
    throws Exception {
        return integer + integer2;
    }
}).collect();
// finally we sort the result. Note we need to create a
// Comparator function,
// that reverses the sort order.
Collections.sort(pairs, new Comparator<Tuple2<String, Integer>>() {
    @Override
    public int compare(Tuple2<String, Integer> o1,
            Tuple2<String, Integer> o2) {
        return -(o1._2() - o2._2());
    }
});
String mostPopular = pairs.get(0)._1();
int purchases = pairs.get(0)._2();
System.out.println("Total purchases: "+numPurchases);
System.out.println("Unique users: "+uniqueUsers);
System.out.println("Total revenue: "+totalRevenue);
System.out.println(String.format("Most popular product: 
%s with %d purchases", mostPopular, purchases));
```
As can be seen, the general structure is similar to the Scala version, apart from the extra boilerplate code to declare variables and functions via anonymous inner classes. It is a good exercise to work through both examples and compare the same lines of Scala code to those in Java to understand how the same result is achieved in each language.

This program can be run with the following command executed from the project's base directory:

```
>mvn exec:java -Dexec.mainClass="JavaApp"
```

You will see output that looks very similar to the Scala version, with the results of the computation identical:

```
...
14/01/30 17:02:43 INFO spark.SparkContext: Job finished: collect at JavaApp.java:46, took 0.039167 s
Total purchases: 5
Unique users: 4
Total revenue: 39.91
Most popular product: iPhone Cover with 2 purchases
```

The first step to a Spark program in Python

Spark's Python API exposes virtually all the functionalities of Spark's Scala API in the Python language. There are some features that are not yet supported (for example, graph processing with GraphX and a few API methods here and there). See the Python section of the Spark Programming Guide (http://spark.apache.org/docs/latest/programming-guide.html) for more details.

Following on from the preceding examples, we will now write a Python version. We assume that you have Python version 2.6 and higher installed on your system (for example, most Linux and Mac OS X systems come with Python preinstalled).

The example program is included in the sample code for this chapter, in the directory named python-spark-app, which also contains the CSV data file under the data subdirectory. The project contains a script, pythonapp.py, provided here:

```
"""A simple Spark app in Python"

from pyspark import SparkContext
```
sc = SparkContext("local[2]", "First Spark App")
# we take the raw data in CSV format and convert it into a set of
# records of the form (user, product, price)
data = sc.textFile("data/UserPurchaseHistory.csv").map(lambda line:
    line.split(",")).map(lambda record: (record[0], record[1], record[2])))
# let's count the number of purchases
numPurchases = data.count()
# let's count how many unique users made purchases
uniqueUsers = data.map(lambda record: record[0]).distinct().count()
# let's sum up our total revenue
totalRevenue = data.map(lambda record: float(record[2])).sum()
# let's find our most popular product
products = data.map(lambda record: (record[1], 1.0)).
    reduceByKey(lambda a, b: a + b).collect()
mostPopular = sorted(products, key=lambda x: x[1], reverse=True)[0]

print "Total purchases: %d" % numPurchases
print "Unique users: %d" % uniqueUsers
print "Total revenue: %2.2f" % totalRevenue
print "Most popular product: %s with %d purchases" % (mostPopular[0],
    mostPopular[1])

If you compare the Scala and Python versions of our program, you will see that
generally, the syntax looks very similar. One key difference is how we express
anonymous functions (also called lambda functions; hence, the use of this keyword
for the Python syntax). In Scala, we've seen that an anonymous function mapping an
input x to an output y is expressed as x => y, while in Python, it is lambda x: y. In
the highlighted line in the preceding code, we are applying an anonymous function
that maps two inputs, a and b, generally of the same type, to an output. In this case,
the function that we apply is the plus function; hence, lambda a, b: a + b.

The best way to run the script is to run the following command from the base
directory of the sample project:

> $SPARK_HOME/bin/spark-submit pythonapp.py

Here, the SPARK_HOME variable should be replaced with the path of the directory
in which you originally unpacked the Spark prebuilt binary package at the start of
this chapter.

Upon running the script, you should see output similar to that of the Scala and Java
examples, with the results of our computation being the same:

...
Getting Spark running on Amazon EC2

The Spark project provides scripts to run a Spark cluster in the cloud on Amazon's EC2 service. These scripts are located in the ec2 directory. You can run the spark-ec2 script contained in this directory with the following command:

```
>./ec2/spark-ec2
```

Running it in this way without an argument will show the help output:

**Usage:** spark-ec2 [options] <action> <cluster_name>

<action> can be: launch, destroy, login, stop, start, get-master

**Options:**

... 

Before creating a Spark EC2 cluster, you will need to ensure you have an Amazon account.

If you don't have an Amazon Web Services account, you can sign up at http://aws.amazon.com/.
The AWS console is available at http://aws.amazon.com/console/.

You will also need to create an Amazon EC2 key pair and retrieve the relevant security credentials. The Spark documentation for EC2 (available at http://spark.apache.org/docs/latest/ec2-scripts.html) explains the requirements:

Create an Amazon EC2 key pair for yourself. This can be done by logging into your Amazon Web Services account through the AWS console, clicking on **Key Pairs** on the left sidebar, and creating and downloading a key. Make sure that you set the permissions for the private key file to 600 (that is, only you can read and write it) so that ssh will work.

Whenever you want to use the spark-ec2 script, set the environment variables AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY to your Amazon EC2 access key ID and secret access key, respectively. These can be obtained from the AWS homepage by clicking **Account** | **Security Credentials** | **Access Credentials**.
When creating a key pair, choose a name that is easy to remember. We will simply use spark for the key pair name. The key pair file itself will be called spark.pem. As mentioned earlier, ensure that the key pair file permissions are set appropriately and that the environment variables for the AWS credentials are exported using the following commands:

```
> chmod 600 spark.pem
> export AWS_ACCESS_KEY_ID="..."
> export AWS_SECRET_ACCESS_KEY="..."
```

You should also be careful to keep your downloaded key pair file safe and not lose it, as it can only be downloaded once when it is created!

Note that launching an Amazon EC2 cluster in the following section will incur costs to your AWS account.

## Launching an EC2 Spark cluster

We're now ready to launch a small Spark cluster by changing into the ec2 directory and then running the cluster launch command:

```
> cd ec2
> ./spark-ec2 -k spark -i spark.pem -s 1 --instance-type m3.medium --hadoop-major-version 2 launch test-cluster
```

This will launch a new Spark cluster called test-cluster with one master and one slave node of instance type m3.medium. This cluster will be launched with a Spark version built for Hadoop 2. The key pair name we used is spark, and the key pair file is spark.pem (if you gave the files different names or have an existing AWS key pair, use that name instead).

It might take quite a while for the cluster to fully launch and initialize. You should see something like this screenshot immediately after running the launch command:
If the cluster has launched successfully, you should eventually see the console output similar to the following screenshot:

```
- Amazon Linux AMI

There are 68 security update(s) out of 254 total update(s) available
Run "sudo yum update" to apply all updates.
Amazon Linux version 2014.09 is available.
root@ip-10-0-14-70:~# 15
```

To test whether we can connect to our new cluster, we can run the following command:

```
> ssh -i spark.pem root@ec2-54-227-127-14.compute-1.amazonaws.com
```

Remember to replace the public domain name of the master node (the address after `root@` in the preceding command) with the correct Amazon EC2 public domain name that will be shown in your console output after launching the cluster.

You can also retrieve your cluster’s master public domain name by running this line of code:

```
> ./spark-ec2 -i spark.pem get-master test-cluster
```

After successfully running the `ssh` command, you will be connected to your Spark master node in EC2, and your terminal output should match the following screenshot:
We can test whether our cluster is correctly set up with Spark by changing into the Spark directory and running an example in the local mode:

```
> cd spark
> MASTER=local[2] ./bin/run-example SparkPi
```

You should see output similar to running the same command on your local computer:

```
...
14/01/30 20:20:21 INFO SparkContext: Job finished: reduce at SparkPi.
scala:35, took 0.864044012 s
Pi is roughly 3.14032
...
```

Now that we have an actual cluster with multiple nodes, we can test Spark in the cluster mode. We can run the same example on the cluster, using our 1 slave node, by passing in the master URL instead of the local version:

```
> MASTER=spark://ec2-54-227-127-14.compute-1.amazonaws.com:7077 ./bin/run-example SparkPi
```

Note that you will need to substitute the preceding master domain name with the correct domain name for your specific cluster.

Again, the output should be similar to running the example locally; however, the log messages will show that your driver program has connected to the Spark master:

```
...
14/01/30 20:26:17 INFO cluster.SparkDeploySchedulerBackend: Connected to Spark cluster with app ID app-20140130202617-0001
14/01/30 20:26:17 INFO client.Client$ClientActor: Executor added: app-20140130202617-0001/0 on worker-20140130201049-ip-10-34-137-45.eu-west-1.compute.internal-57119 (ip-10-34-137-45.eu-west-1.compute.internal:57119) with 1 cores
14/01/30 20:26:17 INFO cluster.SparkDeploySchedulerBackend: Granted executor ID app-20140130202617-0001/0 on hostPort ip-10-34-137-45.eu-west-1.compute.internal:57119 with 1 cores, 2.4 GB RAM
```
Getting Up and Running with Spark

Feel free to experiment with your cluster. Try out the interactive console in Scala, for example:

> ./bin/spark-shell --master spark://ec2-54-227-127-14.compute-1.amazonaws.com:7077

Once you've finished, type exit to leave the console. You can also try the PySpark console by running the following command:

> ./bin/pyspark --master spark://ec2-54-227-127-14.compute-1.amazonaws.com:7077

You can use the Spark Master web interface to see the applications registered with the master. To load the Master Web UI, navigate to ec2-54-227-127-14.compute-1.amazonaws.com:8080 (again, remember to replace this domain name with your own master domain name). You should see something similar to the following screenshot showing the example you ran as well as the two console applications you launched:
Remember that you will be charged by Amazon for usage of the cluster. Don't forget to stop or terminate this test cluster once you're done with it. To do this, you can first exit the ssh session by typing `exit` to return to your own local system and then, run the following command:

```
>./ec2/spark-ec2 -k spark -i spark.pem destroy test-cluster
```

You should see the following output:

Are you sure you want to destroy the cluster test-cluster?
The following instances will be terminated:
Searching for existing cluster test-cluster...
Found 1 master(s), 1 slaves
  > ec2-54-227-127-14.compute-1.amazonaws.com
  > ec2-54-91-61-225.compute-1.amazonaws.com
ALL DATA ON ALL NODES WILL BE LOST!!
Destroy cluster test-cluster (y/N): y
Terminating master...
Terminating slaves...

Hit Y and then Enter to destroy the cluster.

Congratulations! You've just set up a Spark cluster in the cloud, run a fully parallel example program on this cluster, and terminated it. If you would like to try out any of the example code in the subsequent chapters (or your own Spark programs) on a cluster, feel free to experiment with the Spark EC2 scripts and launch a cluster of your chosen size and instance profile (just be mindful of the costs and remember to shut it down when you're done!).

**Summary**

In this chapter, we covered how to set up Spark locally on our own computer as well as in the cloud as a cluster running on Amazon EC2. You learned the basics of Spark's programming model and API using the interactive Scala console, and we wrote the same basic Spark program in Scala, Java, and Python.

In the next chapter, we will consider how to go about using Spark to create a machine learning system.
Where to buy this book

You can buy Machine Learning with Spark from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.