Foreword

In a very short time, Apache Spark has emerged as the next generation big data processing engine, and is being applied throughout the industry faster than ever. Spark improves over Hadoop MapReduce, which helped ignite the big data revolution, in several key dimensions: it is much faster, much easier to use due to its rich APIs, and it goes far beyond batch applications to support a variety of workloads, including interactive queries, streaming, machine learning, and graph processing.

I have been privileged to be closely involved with the development of Spark all the way from the drawing board to what has become the most active big data open source project today, and one of the most active Apache projects! As such, I'm particularly delighted to see Matei Zaharia, the creator of Spark, teaming up with other longtime Spark developers Patrick Wendell, Andy Konwinski, and Holden Karau to write this book.

With Spark's rapid rise in popularity, a major concern has been lack of good reference material. This book goes a long way to address this concern, with 11 chapters and dozens of detailed examples designed for data scientists, students, and developers looking to learn Spark. It is written to be approachable by readers with no background in big data, making it a great place to start learning about the field in general. I hope that many years from now, you and other readers will fondly remember this as *the* book that introduced you to this exciting new field.

—Ion Stoica, CEO of Databricks and Co-director, AMPlab, UC Berkeley

Preface

As parallel data analysis has grown common, practitioners in many fields have sought easier tools for this task. Apache Spark has quickly emerged as one of the most popular, extending and generalizing MapReduce. Spark offers three main benefits. First, it is easy to use—you can develop applications on your laptop, using a high-level API that lets you focus on the content of your computation. Second, Spark is fast, enabling interactive use and complex algorithms. And third, Spark is a *general* engine, letting you combine multiple types of computations (e.g., SQL queries, text processing, and machine learning) that might previously have required different engines. These features make Spark an excellent starting point to learn about Big Data in general.

This introductory book is meant to get you up and running with Spark quickly. You'll learn how to download and run Spark on your laptop and use it interactively to learn the API. Once there, we'll cover the details of available operations and distributed execution. Finally, you'll get a tour of the higher-level libraries built into Spark, including libraries for machine learning, stream processing, and SQL. We hope that this book gives you the tools to quickly tackle data analysis problems, whether you do so on one machine or hundreds.

Audience

This book targets data scientists and engineers. We chose these two groups because they have the most to gain from using Spark to expand the scope of problems they can solve. Spark's rich collection of data-focused libraries (like MLlib) makes it easy for data scientists to go beyond problems that fit on a single machine while using their statistical background. Engineers, meanwhile, will learn how to write general-purpose distributed programs in Spark and operate production applications. Engineers and data scientists will both learn different details from this book, but will both be able to apply Spark to solve large distributed problems in their respective fields.

Data scientists focus on answering questions or building models from data. They often have a statistical or math background and some familiarity with tools like Python, R, and SQL. We have made sure to include Python and, where relevant, SQL examples for all our material, as well as an overview of the machine learning and library in Spark. If you are a data scientist, we hope that after reading this book you will be able to use the same mathematical approaches to solve problems, except much faster and on a much larger scale.

The second group this book targets is software engineers who have some experience with Java, Python, or another programming language. If you are an engineer, we hope that this book will show you how to set up a Spark cluster, use the Spark shell, and write Spark applications to solve parallel processing problems. If you are familiar with Hadoop, you have a bit of a head start on figuring out how to interact with HDFS and how to manage a cluster, but either way, we will cover basic distributed execution concepts.

Regardless of whether you are a data scientist or engineer, to get the most out of this book you should have some familiarity with one of Python, Java, Scala, or a similar language. We assume that you already have a storage solution for your data and we cover how to load and save data from many common ones, but not how to set them up. If you don't have experience with one of those languages, don't worry: there are excellent resources available to learn these. We call out some of the books available in "Supporting Books" on page xii.

How This Book Is Organized

The chapters of this book are laid out in such a way that you should be able to go through the material front to back. At the start of each chapter, we will mention which sections we think are most relevant to data scientists and which sections we think are most relevant for engineers. That said, we hope that all the material is accessible to readers of either background.

The first two chapters will get you started with getting a basic Spark installation on your laptop and give you an idea of what you can accomplish with Spark. Once we've got the motivation and setup out of the way, we will dive into the Spark shell, a very useful tool for development and prototyping. Subsequent chapters then cover the Spark programming interface in detail, how applications execute on a cluster, and higher-level libraries available on Spark (such as Spark SQL and MLlib).

Supporting Books

If you are a data scientist and don't have much experience with Python, the books *Learning Python* and *Head First Python* (both O'Reilly) are excellent introductions. If you have some Python experience and want more, *Dive into Python* (Apress) is a great book to help you get a deeper understanding of Python.

If you are an engineer and after reading this book you would like to expand your data analysis skills, *Machine Learning for Hackers* and *Doing Data Science* are excellent books (both O'Reilly).

This book is intended to be accessible to beginners. We do intend to release a deep-dive follow-up for those looking to gain a more thorough understanding of Spark's internals.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip or suggestion.

This element indicates a warning or caution.

Code Examples

All of the code examples found in this book are on GitHub. You can examine them and check them out from *https://github.com/databricks/learning-spark*. Code examples are provided in Java, Scala, and Python.

Our Java examples are written to work with Java version 6 and higher. Java 8 introduces a new syntax called *lambdas* that makes writing inline functions much easier, which can simplify Spark code. We have chosen not to take advantage of this syntax in most of our examples, as most organizations are not yet using Java 8. If you would like to try Java 8 syntax, you can see the Databricks blog post on this topic. Some of the examples will also be ported to Java 8 and posted to the book's GitHub site.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Learning Spark* by Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia (O'Reilly). Copyright 2015 Databricks, 978-1-449-35862-4."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of plans and pricing for enterprise, government, education, and individuals.

Members have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan

Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and hundreds more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://bit.ly/learning-spark*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

The authors would like to thank the reviewers who offered feedback on this book: Joseph Bradley, Dave Bridgeland, Chaz Chandler, Mick Davies, Sam DeHority, Vida Ha, Andrew Gal, Michael Gregson, Jan Joeppen, Stephan Jou, Jeff Martinez, Josh Mahonin, Andrew Or, Mike Patterson, Josh Rosen, Bruce Szalwinski, Xiangrui Meng, and Reza Zadeh.

The authors would like to extend a special thanks to David Andrzejewski, David Buttler, Juliet Hougland, Marek Kolodziej, Taka Shinagawa, Deborah Siegel, Dr. Normen Müller, Ali Ghodsi, and Sameer Farooqui. They provided detailed feedback on the majority of the chapters and helped point out many significant improvements.

We would also like to thank the subject matter experts who took time to edit and write parts of their own chapters. Tathagata Das worked with us on a very tight schedule to finish Chapter 10. Tathagata went above and beyond with clarifying

examples, answering many questions, and improving the flow of the text in addition to his technical contributions. Michael Armbrust helped us check the Spark SQL chapter for correctness. Joseph Bradley provided the introductory example for MLlib in Chapter 11. Reza Zadeh provided text and code examples for dimensionality reduction. Xiangrui Meng, Joseph Bradley, and Reza Zadeh also provided editing and technical feedback for the MLlib chapter.

CHAPTER 1

Introduction to Data Analysis with Spark

This chapter provides a high-level overview of what Apache Spark is. If you are already familiar with Apache Spark and its components, feel free to jump ahead to Chapter 2.

What Is Apache Spark?

Apache Spark is a cluster computing platform designed to be *fast* and *general-purpose*.

On the speed side, Spark extends the popular MapReduce model to efficiently support more types of computations, including interactive queries and stream processing. Speed is important in processing large datasets, as it means the difference between exploring data interactively and waiting minutes or hours. One of the main features Spark offers for speed is the ability to run computations in memory, but the system is also more efficient than MapReduce for complex applications running on disk.

On the generality side, Spark is designed to cover a wide range of workloads that previously required separate distributed systems, including batch applications, iterative algorithms, interactive queries, and streaming. By supporting these workloads in the same engine, Spark makes it easy and inexpensive to *combine* different processing types, which is often necessary in production data analysis pipelines. In addition, it reduces the management burden of maintaining separate tools.

Spark is designed to be highly accessible, offering simple APIs in Python, Java, Scala, and SQL, and rich built-in libraries. It also integrates closely with other Big Data tools. In particular, Spark can run in Hadoop clusters and access any Hadoop data source, including Cassandra.

A Unified Stack

The Spark project contains multiple closely integrated components. At its core, Spark is a "computational engine" that is responsible for scheduling, distributing, and monitoring applications consisting of many computational tasks across many worker machines, or a *computing cluster*. Because the core engine of Spark is both fast and general-purpose, it powers multiple higher-level components specialized for various workloads, such as SQL or machine learning. These components are designed to interoperate closely, letting you combine them like libraries in a software project.

A philosophy of tight integration has several benefits. First, all libraries and higher-level components in the stack benefit from improvements at the lower layers. For example, when Spark's core engine adds an optimization, SQL and machine learning libraries automatically speed up as well. Second, the costs associated with running the stack are minimized, because instead of running 5–10 independent software systems, an organization needs to run only one. These costs include deployment, maintenance, testing, support, and others. This also means that each time a new component is added to the Spark stack, every organization that uses Spark will immediately be able to try this new component. This changes the cost of trying out a new type of data analysis from downloading, deploying, and learning a new software project to upgrading Spark.

Finally, one of the largest advantages of tight integration is the ability to build applications that seamlessly combine different processing models. For example, in Spark you can write one application that uses machine learning to classify data in real time as it is ingested from streaming sources. Simultaneously, analysts can query the resulting data, also in real time, via SQL (e.g., to join the data with unstructured logfiles). In addition, more sophisticated data engineers and data scientists can access the same data via the Python shell for ad hoc analysis. Others might access the data in standalone batch applications. All the while, the IT team has to maintain only one system.

Here we will briefly introduce each of Spark's components, shown in Figure 1-1.

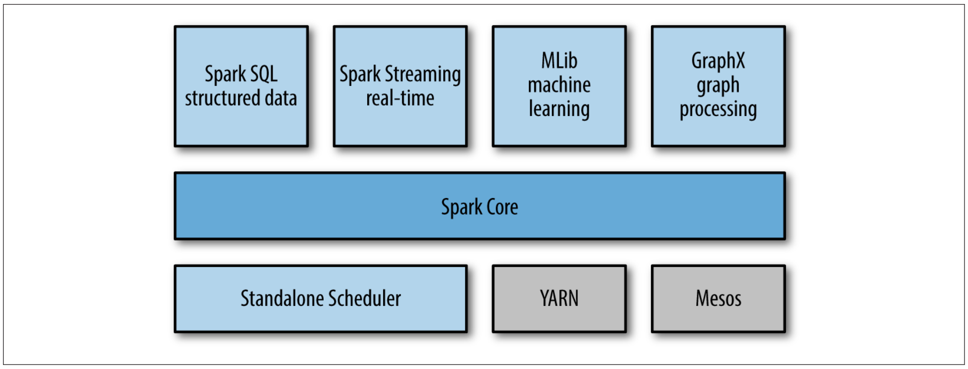


Figure 1-1. The Spark stack

Spark Core

Spark Core contains the basic functionality of Spark, including components for task scheduling, memory management, fault recovery, interacting with storage systems, and more. Spark Core is also home to the API that defines *resilient distributed datasets* (RDDs), which are Spark's main programming abstraction. RDDs represent a collection of items distributed across many compute nodes that can be manipulated in parallel. Spark Core provides many APIs for building and manipulating these collections.

Spark SQL

Spark SQL is Spark's package for working with structured data. It allows querying data via SQL as well as the Apache Hive variant of SQL—called the Hive Query Language (HQL)—and it supports many sources of data, including Hive tables, Parquet, and JSON. Beyond providing a SQL interface to Spark, Spark SQL allows developers to intermix SQL queries with the programmatic data manipulations supported by RDDs in Python, Java, and Scala, all within a single application, thus combining SQL with complex analytics. This tight integration with the rich computing environment provided by Spark makes Spark SQL unlike any other open source data warehouse tool. Spark SQL was added to Spark in version 1.0.

Shark was an older SQL-on-Spark project out of the University of California, Berkeley, that modified Apache Hive to run on Spark. It has now been replaced by Spark SQL to provide better integration with the Spark engine and language APIs.

Spark Streaming

Spark Streaming is a Spark component that enables processing of live streams of data. Examples of data streams include logfiles generated by production web servers, or queues of messages containing status updates posted by users of a web service. Spark

Streaming provides an API for manipulating data streams that closely matches the Spark Core's RDD API, making it easy for programmers to learn the project and move between applications that manipulate data stored in memory, on disk, or arriving in real time. Underneath its API, Spark Streaming was designed to provide the same degree of fault tolerance, throughput, and scalability as Spark Core.

MLlib

Spark comes with a library containing common machine learning (ML) functionality, called MLlib. MLlib provides multiple types of machine learning algorithms, including classification, regression, clustering, and collaborative filtering, as well as supporting functionality such as model evaluation and data import. It also provides some lower-level ML primitives, including a generic gradient descent optimization algorithm. All of these methods are designed to scale out across a cluster.

GraphX

GraphX is a library for manipulating graphs (e.g., a social network's friend graph) and performing graph-parallel computations. Like Spark Streaming and Spark SQL, GraphX extends the Spark RDD API, allowing us to create a directed graph with arbitrary properties attached to each vertex and edge. GraphX also provides various operators for manipulating graphs (e.g., `subgraph` and `mapVertices`) and a library of common graph algorithms (e.g., PageRank and triangle counting).

Cluster Managers

Under the hood, Spark is designed to efficiently scale up from one to many thousands of compute nodes. To achieve this while maximizing flexibility, Spark can run over a variety of *cluster managers*, including Hadoop YARN, Apache Mesos, and a simple cluster manager included in Spark itself called the Standalone Scheduler. If you are just installing Spark on an empty set of machines, the Standalone Scheduler provides an easy way to get started; if you already have a Hadoop YARN or Mesos cluster, however, Spark's support for these cluster managers allows your applications to also run on them. Chapter 7 explores the different options and how to choose the correct cluster manager.

Who Uses Spark, and for What?

Because Spark is a general-purpose framework for cluster computing, it is used for a diverse range of applications. In the Preface we outlined two groups of readers that this book targets: data scientists and engineers. Let's take a closer look at each group and how it uses Spark. Unsurprisingly, the typical use cases differ between the two, but we can roughly classify them into two categories, *data science* and *data applications*.

Of course, these are imprecise disciplines and usage patterns, and many folks have skills from both, sometimes playing the role of the investigating data scientist, and then "changing hats" and writing a hardened data processing application. Nonetheless, it can be illuminating to consider the two groups and their respective use cases separately.

Data Science Tasks

Data science, a discipline that has been emerging over the past few years, centers on analyzing data. While there is no standard definition, for our purposes a *data scientist* is somebody whose main task is to analyze and model data. Data scientists may have experience with SQL, statistics, predictive modeling (machine learning), and programming, usually in Python, Matlab, or R. Data scientists also have experience with techniques necessary to transform data into formats that can be analyzed for insights (sometimes referred to as *data wrangling*).

Data scientists use their skills to analyze data with the goal of answering a question or discovering insights. Oftentimes, their workflow involves ad hoc analysis, so they use interactive shells (versus building complex applications) that let them see results of queries and snippets of code in the least amount of time. Spark's speed and simple APIs shine for this purpose, and its built-in libraries mean that many algorithms are available out of the box.

Spark supports the different tasks of data science with a number of components. The Spark shell makes it easy to do interactive data analysis using Python or Scala. Spark SQL also has a separate SQL shell that can be used to do data exploration using SQL, or Spark SQL can be used as part of a regular Spark program or in the Spark shell. Machine learning and data analysis is supported through the MLLib libraries. In addition, there is support for calling out to external programs in Matlab or R. Spark enables data scientists to tackle problems with larger data sizes than they could before with tools like R or Pandas.

Sometimes, after the initial exploration phase, the work of a data scientist will be "productized," or extended, hardened (i.e., made fault-tolerant), and tuned to become a production data processing application, which itself is a component of a business application. For example, the initial investigation of a data scientist might lead to the creation of a production recommender system that is integrated into a web application and used to generate product suggestions to users. Often it is a different person or team that leads the process of productizing the work of the data scientists, and that person is often an engineer.

Data Processing Applications

The other main use case of Spark can be described in the context of the engineer persona. For our purposes here, we think of engineers as a large class of software developers who use Spark to build production data processing applications. These developers usually have an understanding of the principles of software engineering, such as encapsulation, interface design, and object-oriented programming. They frequently have a degree in computer science. They use their engineering skills to design and build software systems that implement a business use case.

For engineers, Spark provides a simple way to parallelize these applications across clusters, and hides the complexity of distributed systems programming, network communication, and fault tolerance. The system gives them enough control to monitor, inspect, and tune applications while allowing them to implement common tasks quickly. The modular nature of the API (based on passing distributed collections of objects) makes it easy to factor work into reusable libraries and test it locally.

Spark's users choose to use it for their data processing applications because it provides a wide variety of functionality, is easy to learn and use, and is mature and reliable.

A Brief History of Spark

Spark is an open source project that has been built and is maintained by a thriving and diverse community of developers. If you or your organization are trying Spark for the first time, you might be interested in the history of the project. Spark started in 2009 as a research project in the UC Berkeley RAD Lab, later to become the AMPLab. The researchers in the lab had previously been working on Hadoop MapReduce, and observed that MapReduce was inefficient for iterative and interactive computing jobs. Thus, from the beginning, Spark was designed to be fast for interactive queries and iterative algorithms, bringing in ideas like support for in-memory storage and efficient fault recovery.

Research papers were published about Spark at academic conferences and soon after its creation in 2009, it was already 10–20× faster than MapReduce for certain jobs.

Some of Spark's first users were other groups inside UC Berkeley, including machine learning researchers such as the Mobile Millennium project, which used Spark to monitor and predict traffic congestion in the San Francisco Bay Area. In a very short time, however, many external organizations began using Spark, and today, over 50 organizations list themselves on the Spark PoweredBy page, and dozens speak about their use cases at Spark community events such as Spark Meetups and the Spark Summit. In addition to UC Berkeley, major contributors to Spark include Databricks, Yahoo!, and Intel.

In 2011, the AMPLab started to develop higher-level components on Spark, such as Shark (Hive on Spark)[1] and Spark Streaming. These and other components are sometimes referred to as the Berkeley Data Analytics Stack (BDAS).

Spark was first open sourced in March 2010, and was transferred to the Apache Software Foundation in June 2013, where it is now a top-level project.

Spark Versions and Releases

Since its creation, Spark has been a very active project and community, with the number of contributors growing with each release. Spark 1.0 had over 100 individual contributors. Though the level of activity has rapidly grown, the community continues to release updated versions of Spark on a regular schedule. Spark 1.0 was released in May 2014. This book focuses primarily on Spark 1.1.0 and beyond, though most of the concepts and examples also work in earlier versions.

Storage Layers for Spark

Spark can create distributed datasets from any file stored in the Hadoop distributed filesystem (HDFS) or other storage systems supported by the Hadoop APIs (including your local filesystem, Amazon S3, Cassandra, Hive, HBase, etc.). It's important to remember that Spark does not require Hadoop; it simply has support for storage systems implementing the Hadoop APIs. Spark supports text files, SequenceFiles, Avro, Parquet, and any other Hadoop InputFormat. We will look at interacting with these data sources in Chapter 5.

1 Shark has been replaced by Spark SQL.

CHAPTER 2

Downloading Spark and Getting Started

In this chapter we will walk through the process of downloading and running Spark in local mode on a single computer. This chapter was written for anybody who is new to Spark, including both data scientists and engineers.

Spark can be used from Python, Java, or Scala. To benefit from this book, you don't need to be an expert programmer, but we do assume that you are comfortable with the basic syntax of at least one of these languages. We will include examples in all languages wherever possible.

Spark itself is written in Scala, and runs on the Java Virtual Machine (JVM). To run Spark on either your laptop or a cluster, all you need is an installation of Java 6 or newer. If you wish to use the Python API you will also need a Python interpreter (version 2.6 or newer). Spark does not yet work with Python 3.

Downloading Spark

The first step to using Spark is to download and unpack it. Let's start by downloading a recent precompiled released version of Spark. Visit *http://spark.apache.org/downloads.html*, select the package type of "Pre-built for Hadoop 2.4 and later," and click "Direct Download." This will download a compressed TAR file, or *tarball*, called *spark-1.2.0-bin-hadoop2.4.tgz*.

Windows users may run into issues installing Spark into a directory with a space in the name. Instead, install Spark in a directory with no space (e.g., *C:\spark*).

You don't need to have Hadoop, but if you have an existing Hadoop cluster or HDFS installation, download the matching version. You can do so from *http://spark.apache.org/downloads.html* by selecting a different package type, but they will have slightly different filenames. Building from source is also possible; you can find the latest source code on GitHub or select the package type of "Source Code" when downloading.

Most Unix and Linux variants, including Mac OS X, come with a command-line tool called `tar` that can be used to unpack TAR files. If your operating system does not have the `tar` command installed, try searching the Internet for a free TAR extractor—for example, on Windows, you may wish to try 7-Zip.

Now that we have downloaded Spark, let's unpack it and take a look at what comes with the default Spark distribution. To do that, open a terminal, change to the directory where you downloaded Spark, and untar the file. This will create a new directory with the same name but without the final *.tgz* suffix. Change into that directory and see what's inside. You can use the following commands to accomplish all of that:

```
cd ~
tar -xf spark-1.2.0-bin-hadoop2.4.tgz
cd spark-1.2.0-bin-hadoop2.4
ls
```

In the line containing the `tar` command, the `x` flag tells `tar` we are extracting files, and the `f` flag specifies the name of the tarball. The `ls` command lists the contents of the Spark directory. Let's briefly consider the names and purposes of some of the more important files and directories you see here that come with Spark:

README.md
: Contains short instructions for getting started with Spark.

bin
: Contains executable files that can be used to interact with Spark in various ways (e.g., the Spark shell, which we will cover later in this chapter).

core, streaming, python, …
: Contains the source code of major components of the Spark project.

examples
: Contains some helpful Spark standalone jobs that you can look at and run to learn about the Spark API.

Don't worry about the large number of directories and files the Spark project comes with; we will cover most of these in the rest of this book. For now, let's dive right in and try out Spark's Python and Scala shells. We will start by running some of the examples that come with Spark. Then we will write, compile, and run a simple Spark job of our own.

All of the work we will do in this chapter will be with Spark running in *local mode*; that is, nondistributed mode, which uses only a single machine. Spark can run in a variety of different modes, or environments. Beyond local mode, Spark can also be run on Mesos, YARN, or the Standalone Scheduler included in the Spark distribution. We will cover the various deployment modes in detail in Chapter 7.

Introduction to Spark's Python and Scala Shells

Spark comes with interactive shells that enable ad hoc data analysis. Spark's shells will feel familiar if you have used other shells such as those in R, Python, and Scala, or operating system shells like Bash or the Windows command prompt.

Unlike most other shells, however, which let you manipulate data using the disk and memory on a single machine, Spark's shells allow you to interact with data that is distributed on disk or in memory across many machines, and Spark takes care of automatically distributing this processing.

Because Spark can load data into memory on the worker nodes, many distributed computations, even ones that process terabytes of data across dozens of machines, can run in a few seconds. This makes the sort of iterative, ad hoc, and exploratory analysis commonly done in shells a good fit for Spark. Spark provides both Python and Scala shells that have been augmented to support connecting to a cluster.

Most of this book includes code in all of Spark's languages, but interactive shells are available only in Python and Scala. Because a shell is very useful for learning the API, we recommend using one of these languages for these examples even if you are a Java developer. The API is similar in every language.

The easiest way to demonstrate the power of Spark's shells is to start using one of them for some simple data analysis. Let's walk through the example from the Quick Start Guide in the official Spark documentation.

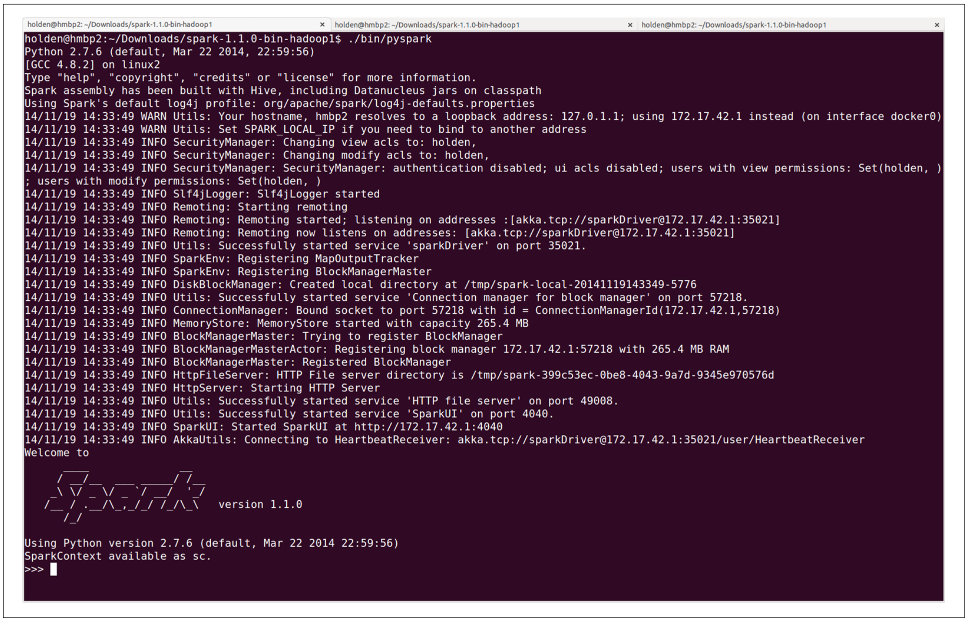
The first step is to open up one of Spark's shells. To open the Python version of the Spark shell, which we also refer to as the PySpark Shell, go into your Spark directory and type:

```
bin/pyspark
```

(Or `bin\pyspark` in Windows.) To open the Scala version of the shell, type:

```
bin/spark-shell
```

The shell prompt should appear within a few seconds. When the shell starts, you will notice a lot of log messages. You may need to press Enter once to clear the log output and get to a shell prompt. Figure 2-1 shows what the PySpark shell looks like when you open it.


holden@hmbp2: ~/Downloads/spark-1.1.0-bin-hadoop1
holden@hmbp2: ~/Downloads/spark-1.1.0-bin-hadoop1
holden@hmbp2: ~/Downloads/spark-1.1.0-bin-hadoop1

```
holden@hmbp2:~/Downloads/spark-1.1.0-bin-hadoop1$ ./bin/pyspark
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
Spark assembly has been built with Hive, including Datanucleus jars on classpath
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
14/11/19 14:33:49 WARN Utils: Your hostname, hmbp2 resolves to a loopback address: 127.0.1.1; using 172.17.42.1 instead (on interface docker0)
14/11/19 14:33:49 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another address
14/11/19 14:33:49 INFO SecurityManager: Changing view acls to: holden,
14/11/19 14:33:49 INFO SecurityManager: Changing modify acls to: holden,
14/11/19 14:33:49 INFO SecurityManager: SecurityManager: authentication disabled; ui acls disabled; users with view permissions: Set(holden, )
; users with modify permissions: Set(holden, )
14/11/19 14:33:49 INFO Slf4jLogger: Slf4jLogger started
14/11/19 14:33:49 INFO Remoting: Starting remoting
14/11/19 14:33:49 INFO Remoting: Remoting started; listening on addresses :[akka.tcp://sparkDriver@172.17.42.1:35021]
14/11/19 14:33:49 INFO Remoting: Remoting now listens on addresses: [akka.tcp://sparkDriver@172.17.42.1:35021]
14/11/19 14:33:49 INFO Utils: Successfully started service 'sparkDriver' on port 35021.
14/11/19 14:33:49 INFO SparkEnv: Registering MapOutputTracker
14/11/19 14:33:49 INFO SparkEnv: Registering BlockManagerMaster
14/11/19 14:33:49 INFO DiskBlockManager: Created local directory at /tmp/spark-local-20141119143349-5776
14/11/19 14:33:49 INFO Utils: Successfully started service 'Connection manager for block manager' on port 57218.
14/11/19 14:33:49 INFO ConnectionManager: Bound socket to port 57218 with id = ConnectionManagerId(172.17.42.1,57218)
14/11/19 14:33:49 INFO MemoryStore: MemoryStore started with capacity 265.4 MB
14/11/19 14:33:49 INFO BlockManagerMaster: Trying to register BlockManager
14/11/19 14:33:49 INFO BlockManagerMasterActor: Registering block manager 172.17.42.1:57218 with 265.4 MB RAM
14/11/19 14:33:49 INFO BlockManagerMaster: Registered BlockManager
14/11/19 14:33:49 INFO HttpFileServer: HTTP File server directory is /tmp/spark-399c53ec-0be8-4043-9a7d-9345e970576d
14/11/19 14:33:49 INFO HttpServer: Starting HTTP Server
14/11/19 14:33:49 INFO Utils: Successfully started service 'HTTP file server' on port 49008.
14/11/19 14:33:49 INFO Utils: Successfully started service 'SparkUI' on port 4040.
14/11/19 14:33:49 INFO SparkUI: Started SparkUI at http://172.17.42.1:4040
14/11/19 14:33:49 INFO AkkaUtils: Connecting to HeartbeatReceiver: akka.tcp://sparkDriver@172.17.42.1:35021/user/HeartbeatReceiver
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 1.1.0
      /_/

Using Python version 2.7.6 (default, Mar 22 2014 22:59:56)
SparkContext available as sc.
>>> 
```

Figure 2-1. The PySpark shell with default logging output

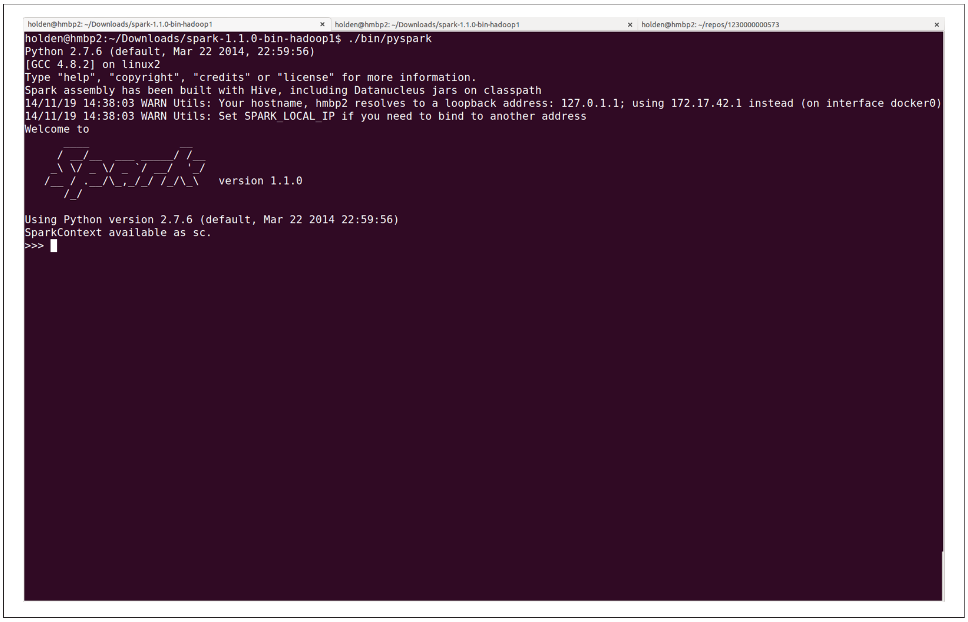
You may find the logging statements that get printed in the shell distracting. You can control the verbosity of the logging. To do this, you can create a file in the *conf* directory called *log4j.properties*. The Spark developers already include a template for this file called *log4j.properties.template*. To make the logging less verbose, make a copy of *conf/log4j.properties.template* called *conf/log4j.properties* and find the following line:

```
log4j.rootCategory=INFO, console
```

Then lower the log level so that we show only the WARN messages, and above by changing it to the following:

```
log4j.rootCategory=WARN, console
```

When you reopen the shell, you should see less output (Figure 2-2).



```
holden@hmbp2:~/Downloads/spark-1.1.0-bin-hadoop1$ ./bin/pyspark
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
Spark assembly has been built with Hive, including Datanucleus jars on classpath
14/11/19 14:38:03 WARN Utils: Your hostname, hmbp2 resolves to a loopback address: 127.0.1.1; using 172.17.42.1 instead (on interface docker0)
14/11/19 14:38:03 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another address
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 1.1.0
      /_/

Using Python version 2.7.6 (default, Mar 22 2014 22:59:56)
SparkContext available as sc.
>>> 
```

holden@hmbp2: ~/Downloads/spark-1.1.0-bin-hadoop1
holden@hmbp2: ~/Downloads/spark-1.1.0-bin-hadoop1
holden@hmbp2: ~/repos/1230000000573

Figure 2-2. The PySpark shell with less logging output

Using IPython

IPython is an enhanced Python shell that many Python users prefer, offering features such as tab completion. You can find instructions for installing it at *http://ipython.org*. You can use IPython with Spark by setting the `IPYTHON` environment variable to 1:

```
IPYTHON=1 ./bin/pyspark
```

To use the IPython Notebook, which is a web-browser-based version of IPython, use:

```
IPYTHON_OPTS="notebook" ./bin/pyspark
```

On Windows, set the variable and run the shell as follows:

```
set IPYTHON=1
bin\pyspark
```

In Spark, we express our computation through operations on distributed collections that are automatically parallelized across the cluster. These collections are called *resilient distributed datasets*, or RDDs. RDDs are Spark's fundamental abstraction for distributed data and computation.

Before we say more about RDDs, let's create one in the shell from a local text file and do some very simple ad hoc analysis by following Example 2-1 for Python or Example 2-2 for Scala.

Example 2-1. Python line count

```
>>> lines = sc.textFile("README.md") # Create an RDD called lines

>>> lines.count() # Count the number of items in this RDD
127
>>> lines.first() # First item in this RDD, i.e. first line of README.md
u'# Apache Spark'
```

Example 2-2. Scala line count

```
scala> val lines = sc.textFile("README.md") // Create an RDD called lines
lines: spark.RDD[String] = MappedRDD[...]

scala> lines.count() // Count the number of items in this RDD
res0: Long = 127

scala> lines.first() // First item in this RDD, i.e. first line of README.md
res1: String = # Apache Spark
```

To exit either shell, press Ctrl-D.

We will discuss it more in Chapter 7, but one of the messages you may have noticed is `INFO SparkUI: Started SparkUI at http://[ipaddress]:4040`. You can access the Spark UI there and see all sorts of information about your tasks and cluster.

In Examples 2-1 and 2-2, the variable called `lines` is an RDD, created here from a text file on our local machine. We can run various parallel operations on the RDD, such as counting the number of elements in the dataset (here, lines of text in the file) or printing the first one. We will discuss RDDs in great depth in later chapters, but before we go any further, let's take a moment now to introduce basic Spark concepts.

Introduction to Core Spark Concepts

Now that you have run your first Spark code using the shell, it's time to learn about programming in it in more detail.

At a high level, every Spark application consists of a *driver program* that launches various parallel operations on a cluster. The driver program contains your application's `main` function and defines distributed datasets on the cluster, then applies operations to them. In the preceding examples, the driver program was the Spark shell itself, and you could just type in the operations you wanted to run.

Driver programs access Spark through a `SparkContext` object, which represents a connection to a computing cluster. In the shell, a SparkContext is automatically created for you as the variable called sc. Try printing out sc to see its type, as shown in Example 2-3.

Example 2-3. Examining the sc variable

```
>>> sc
<pyspark.context.SparkContext object at 0x1025b8f90>
```

Once you have a SparkContext, you can use it to build RDDs. In Examples 2-1 and 2-2, we called `sc.textFile()` to create an RDD representing the lines of text in a file. We can then run various operations on these lines, such as `count()`.

To run these operations, driver programs typically manage a number of nodes called *executors*. For example, if we were running the `count()` operation on a cluster, different machines might count lines in different ranges of the file. Because we just ran the Spark shell locally, it executed all its work on a single machine—but you can connect the same shell to a cluster to analyze data in parallel. Figure 2-3 shows how Spark executes on a cluster.

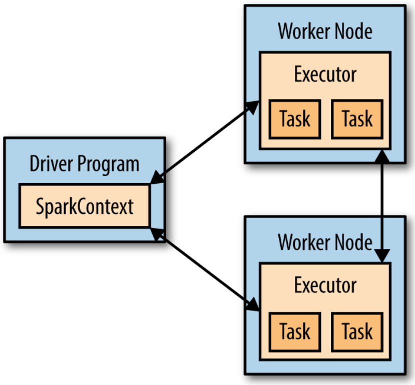


Figure 2-3. Components for distributed execution in Spark

Finally, a lot of Spark's API revolves around passing functions to its operators to run them on the cluster. For example, we could extend our README example by *filtering* the lines in the file that contain a word, such as *Python*, as shown in Example 2-4 (for Python) and Example 2-5 (for Scala).

Example 2-4. Python filtering example

```
>>> lines = sc.textFile("README.md")

>>> pythonLines = lines.filter(lambda line: "Python" in line)
>>> pythonLines.first()
u'## Interactive Python Shell'
```

Example 2-5. Scala filtering example

```
scala> val lines = sc.textFile("README.md") // Create an RDD called lines
lines: spark.RDD[String] = MappedRDD[...]

scala> val pythonLines = lines.filter(line => line.contains("Python"))
pythonLines: spark.RDD[String] = FilteredRDD[...]

scala> pythonLines.first()
res0: String = ## Interactive Python Shell
```

Passing Functions to Spark

If you are unfamiliar with the `lambda` or `=>` syntax in Examples 2-4 and 2-5, it is a shorthand way to define functions inline in Python and Scala. When using Spark in these languages, you can also define a function separately and then pass its name to Spark. For example, in Python:

```
def hasPython(line):
    return "Python" in line

pythonLines = lines.filter(hasPython)
```

Passing functions to Spark is also possible in Java, but in this case they are defined as classes, implementing an interface called `Function`. For example:

```
JavaRDD<String> pythonLines = lines.filter(
  new Function<String, Boolean>() {
    Boolean call(String line) { return line.contains("Python"); }
  }
);
```

Java 8 introduces shorthand syntax called *lambdas* that looks similar to Python and Scala. Here is how the code would look with this syntax:

```
JavaRDD<String> pythonLines = lines.filter(line -> line.contains("Python"));
```

We discuss passing functions further in "Passing Functions to Spark" on page 30.

While we will cover the Spark API in more detail later, a lot of its magic is that function-based operations like `filter` *also* parallelize across the cluster. That is, Spark automatically takes your function (e.g., `line.contains("Python")`) and ships it to executor nodes. Thus, you can write code in a single driver program and automatically have parts of it run on multiple nodes. Chapter 3 covers the RDD API in detail.

Standalone Applications

The final piece missing in this quick tour of Spark is how to use it in standalone programs. Apart from running interactively, Spark can be linked into standalone applications in either Java, Scala, or Python. The main difference from using it in the shell is that you need to initialize your own SparkContext. After that, the API is the same.

The process of linking to Spark varies by language. In Java and Scala, you give your application a Maven dependency on the `spark-core` artifact. As of the time of writing, the latest Spark version is 1.2.0, and the Maven coordinates for that are:

```
groupId = org.apache.spark
artifactId = spark-core_2.10
version = 1.2.0
```

Maven is a popular package management tool for Java-based languages that lets you link to libraries in public repositories. You can use Maven itself to build your project, or use other tools that can talk to the Maven repositories, including Scala's sbt tool or Gradle. Popular integrated development environments like Eclipse also allow you to directly add a Maven dependency to a project.

In Python, you simply write applications as Python scripts, but you must run them using the `bin/spark-submit` script included in Spark. The `spark-submit` script includes the Spark dependencies for us in Python. This script sets up the environment for Spark's Python API to function. Simply run your script with the line given in Example 2-6.

Example 2-6. Running a Python script

```
bin/spark-submit my_script.py
```

(Note that you will have to use backslashes instead of forward slashes on Windows.)

Initializing a SparkContext

Once you have linked an application to Spark, you need to import the Spark packages in your program and create a SparkContext. You do so by first creating a `SparkConf` object to configure your application, and then building a SparkContext for it. Examples 2-7 through 2-9 demonstrate this in each supported language.

Example 2-7. Initializing Spark in Python

```python
from pyspark import SparkConf, SparkContext

conf = SparkConf().setMaster("local").setAppName("My App")
sc = SparkContext(conf = conf)
```

Example 2-8. Initializing Spark in Scala

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

val conf = new SparkConf().setMaster("local").setAppName("My App")
val sc = new SparkContext(conf)
```

Example 2-9. Initializing Spark in Java

```
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaSparkContext;

SparkConf conf = new SparkConf().setMaster("local").setAppName("My App");
JavaSparkContext sc = new JavaSparkContext(conf);
```

These examples show the minimal way to initialize a SparkContext, where you pass two parameters:

- A *cluster URL*, namely `local` in these examples, which tells Spark how to connect to a cluster. `local` is a special value that runs Spark on one thread on the local machine, without connecting to a cluster.
- An *application name*, namely `My App` in these examples. This will identify your application on the cluster manager's UI if you connect to a cluster.

Additional parameters exist for configuring how your application executes or adding code to be shipped to the cluster, but we will cover these in later chapters of the book.

After you have initialized a SparkContext, you can use all the methods we showed before to create RDDs (e.g., from a text file) and manipulate them.

Finally, to shut down Spark, you can either call the `stop()` method on your SparkContext, or simply exit the application (e.g., with `System.exit(0)` or `sys.exit()`).

This quick overview should be enough to let you run a standalone Spark application on your laptop. For more advanced configuration, Chapter 7 will cover how to connect your application to a cluster, including packaging your application so that its code is automatically shipped to worker nodes. For now, please refer to the Quick Start Guide in the official Spark documentation.

Building Standalone Applications

This wouldn't be a complete introductory chapter of a Big Data book if we didn't have a word count example. On a single machine, implementing word count is simple, but in distributed frameworks it is a common example because it involves reading and combining data from many worker nodes. We will look at building and packaging a simple word count example with both sbt and Maven. All of our examples can be built together, but to illustrate a stripped-down build with minimal dependencies we have a separate smaller project underneath the *learning-spark-examples/mini-complete-example* directory, as you can see in Examples 2-10 (Java) and 2-11 (Scala).

Example 2-10. Word count Java application—don't worry about the details yet

```
// Create a Java Spark Context
SparkConf conf = new SparkConf().setAppName("wordCount");
JavaSparkContext sc = new JavaSparkContext(conf);
// Load our input data.
JavaRDD<String> input = sc.textFile(inputFile);
// Split up into words.
JavaRDD<String> words = input.flatMap(
  new FlatMapFunction<String, String>() {
    public Iterable<String> call(String x) {
      return Arrays.asList(x.split(" "));
    }});
// Transform into pairs and count.
JavaPairRDD<String, Integer> counts = words.mapToPair(
  new PairFunction<String, String, Integer>(){
    public Tuple2<String, Integer> call(String x){
      return new Tuple2(x, 1);
    }}).reduceByKey(new Function2<Integer, Integer, Integer>(){
        public Integer call(Integer x, Integer y){ return x + y;}});
// Save the word count back out to a text file, causing evaluation.
counts.saveAsTextFile(outputFile);
```

Example 2-11. Word count Scala application—don't worry about the details yet

```
// Create a Scala Spark Context.
val conf = new SparkConf().setAppName("wordCount")
val sc = new SparkContext(conf)
// Load our input data.
val input =  sc.textFile(inputFile)
// Split it up into words.
val words = input.flatMap(line => line.split(" "))
// Transform into pairs and count.
val counts = words.map(word => (word, 1)).reduceByKey{case (x, y) => x + y}
// Save the word count back out to a text file, causing evaluation.
counts.saveAsTextFile(outputFile)
```

We can build these applications using very simple build files with both sbt (Example 2-12) and Maven (Example 2-13). We've marked the Spark Core dependency as `provided` so that, later on, when we use an assembly JAR we don't include the `spark-core` JAR, which is already on the classpath of the workers.

Example 2-12. sbt build file

```
name := "learning-spark-mini-example"

version := "0.0.1"

scalaVersion := "2.10.4"

// additional libraries
libraryDependencies ++= Seq(
  "org.apache.spark" %% "spark-core" % "1.2.0" % "provided"
)
```

Example 2-13. Maven build file

```
<project>
  <groupId>com.oreilly.learningsparkexamples.mini</groupId>
  <artifactId>learning-spark-mini-example</artifactId>
  <modelVersion>4.0.0</modelVersion>
  <name>example</name>
  <packaging>jar</packaging>
  <version>0.0.1</version>
  <dependencies>
    <dependency> <!-- Spark dependency -->
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-core_2.10</artifactId>
      <version>1.2.0</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>
  <properties>
    <java.version>1.6</java.version>
  </properties>
  <build>
    <pluginManagement>
      <plugins>
        <plugin>   <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-compiler-plugin</artifactId>
          <version>3.1</version>
          <configuration>
            <source>${java.version}</source>
            <target>${java.version}</target>
          </configuration> </plugin> </plugin>
      </plugins>
    </pluginManagement>
  </build>
</project>
```

The `spark-core` package is marked as `provided` in case we package our application into an assembly JAR. This is covered in more detail in Chapter 7.

Once we have our build defined, we can easily package and run our application using the `bin/spark-submit` script. The `spark-submit` script sets up a number of environment variables used by Spark. From the *mini-complete-example* directory we can build in both Scala (Example 2-14) and Java (Example 2-15).

Example 2-14. Scala build and run

```
sbt clean package
$SPARK_HOME/bin/spark-submit \
  --class com.oreilly.learningsparkexamples.mini.scala.WordCount \
  ./target/...(as above) \
  ./README.md ./wordcounts
```

Example 2-15. Maven build and run

```
mvn clean && mvn compile && mvn package
$SPARK_HOME/bin/spark-submit \
  --class com.oreilly.learningsparkexamples.mini.java.WordCount \
  ./target/learning-spark-mini-example-0.0.1.jar \
  ./README.md ./wordcounts
```

For even more detailed examples of linking applications to Spark, refer to the Quick Start Guide in the official Spark documentation. Chapter 7 covers packaging Spark applications in more detail.

Conclusion

In this chapter, we have covered downloading Spark, running it locally on your laptop, and using it either interactively or from a standalone application. We gave a quick overview of the core concepts involved in programming with Spark: a driver program creates a SparkContext and RDDs, and then runs parallel operations on them. In the next chapter, we will dive more deeply into how RDDs operate.