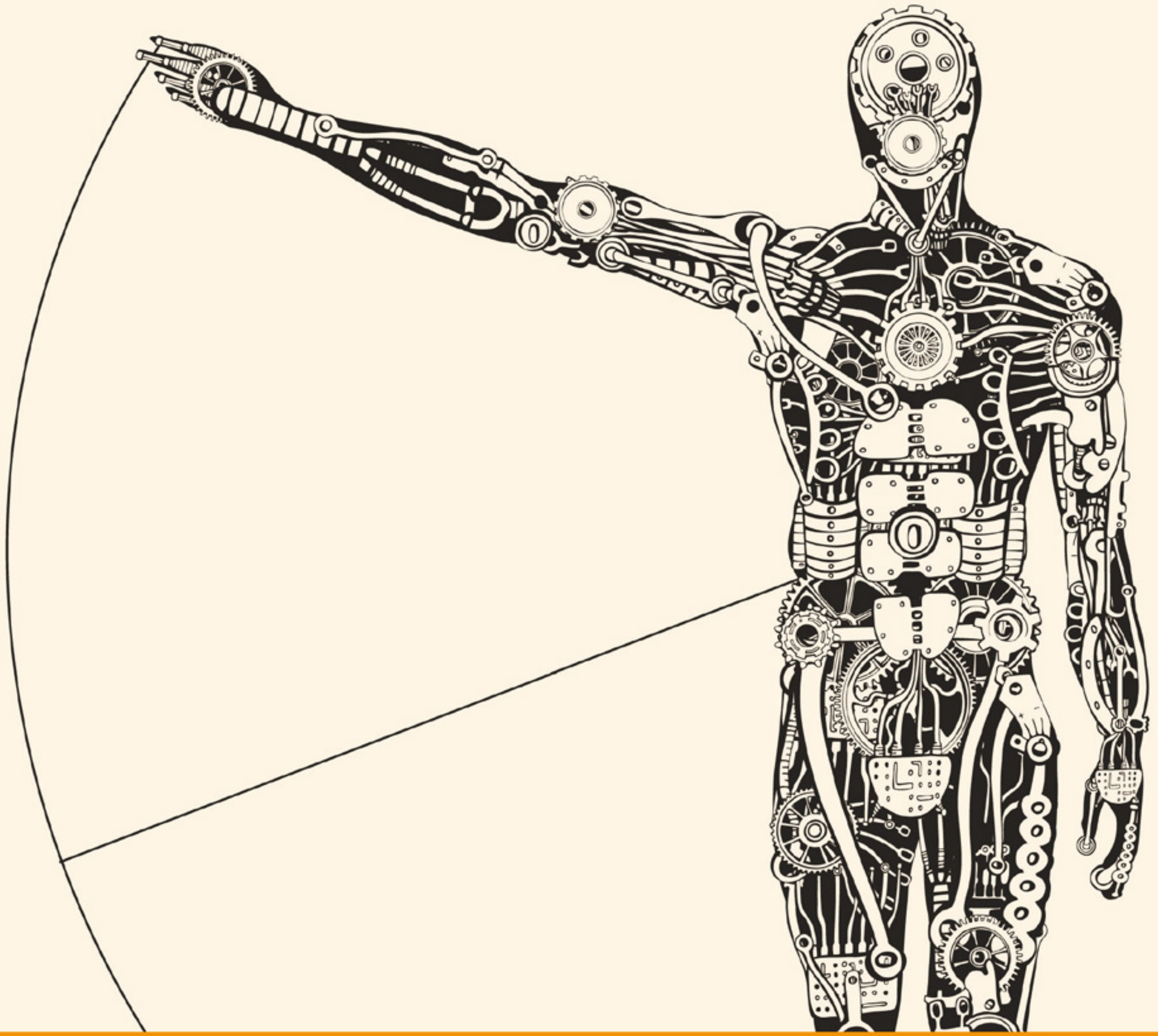


Community Experience Distilled

Fast Data Processing with Spark

Second Edition

Perform real-time analytics using Spark in a fast, distributed, and scalable way

Krishna Sankar
Holden Karau

[PACKT] PUBLISHING open source* community experience distilled

Fast Data Processing with Spark

Second Edition

Perform real-time analytics using Spark in a fast, distributed, and scalable way

Krishna Sankar

Holden Karau

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

Fast Data Processing with Spark
Second Edition

First published: October 2013

Second edition: March 2015

Production reference: 1250315

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78439-257-4

www.packtpub.com

Credits

Authors
Krishna Sankar
Holden Karau

Reviewers
Robin East
Toni Verbeiren
Lijie Xu

Commissioning Editor
Akram Hussain

Acquisition Editors
Shaon Basu
Kunal Parikh

Content Development Editor
Arvind Koul

Technical Editors
Madhunikita Sunil Chindarkar
Taabish Khan

Copy Editor
Hiral Bhat

Project Coordinator
Neha Bhatnagar

Proofreaders
Maria Gould
Ameesha Green
Joanna McMahon

Indexer
Tejal Soni

Production Coordinator
Nilesh R. Mohite

Cover Work
Nilesh R. Mohite

About the Authors

Krishna Sankar is a chief data scientist at `http://www.blackarrow.tv/`, where he focuses on optimizing user experiences via inference, intelligence, and interfaces. His earlier roles include principal architect, data scientist at Tata America Intl, director of a data science and bioinformatics start-up, and a distinguished engineer at Cisco. He has spoken at various conferences, such as Strata-Sparkcamp, OSCON, Pycon, and Pydata about predicting NFL (`http://goo.gl/movfds`), Spark (`http://goo.gl/E4kqMD`), data science (`http://goo.gl/9pyJMH`), machine learning (`http://goo.gl/SXF53n`), and social media analysis (`http://goo.gl/D9YpVQ`). He was a guest lecturer at Naval Postgraduate School, Monterey. His blogs can be found at `https://doubleclix.wordpress.com/`. His other passion is Lego Robotics. You can find him at the St. Louis FLL World Competition as the robots design judge.

The credit goes to my coauthor, Holden Karau, the reviewers, and the editors at Packt Publishing. Holden wrote the first edition, and I hope I was able to contribute to the same depth. I am deeply thankful to the reviewers Lijie, Robin, and Toni. They spent time diligently reviewing the material and code. They have added lots of insightful tips to the text, which I have gratefully included. In addition, their sharp eyes caught tons of errors in the code and text. Thanks to Arvind Koul, who has been the chief force behind the book. A great editor is absolutely essential for the completion of a book, and I was lucky to have Arvind. I also want to thank the editors at Packt Publishing: Anila, Madhunikita, Milton, Neha, and Shaon, with whom I had the fortune to work with at various stages. The guidance and wisdom from Joe Matarese, my boss at `http://www.blackarrow.tv/`, and from Paco Nathan at Databricks are invaluable. My spouse, Usha and son Kaushik, were always with me, cheering me on for any endeavor that I embark upon—mostly successful, like this book, and occasionally foolhardy efforts! I dedicate this book to my mom, who unfortunately passed away last month; she was always proud to see her eldest son as an author.

Holden Karau is a software development engineer and is active in the open source sphere. She has worked on a variety of search, classification, and distributed systems problems at Databricks, Google, Foursquare, and Amazon. She graduated from the University of Waterloo with a bachelor's of mathematics degree in computer science. Other than software, she enjoys playing with fire and hula hoops, and welding.

About the Reviewers

Robin East has served a wide range of roles covering operations research, finance, IT system development, and data science. In the 1980s, he was developing credit scoring models using data science and big data before anyone (including himself) had even heard of those terms! In the last 15 years, he has worked with numerous large organizations, implementing enterprise content search applications, content intelligence systems, and big data processing systems. He has created numerous solutions, ranging from swaps and derivatives in the banking sector to fashion analytics in the retail sector.

Robin became interested in Apache Spark after realizing the limitations of the traditional MapReduce model with respect to running iterative machine learning models. His focus is now on trying to further extend the Spark machine learning libraries, and also on teaching how Spark can be used in data science and data analytics through his blog, Machine Learning at Speed (`http://mlspeed.wordpress.com`).

Before NoSQL databases became the rage, he was an expert on tuning Oracle databases and extracting maximum performance from EMC Documentum systems. This work took him to clients around the world and led him to create the open source profiling tool called DFCprof that is used by hundreds of EMC users to track down performance problems. For many years, he maintained the popular Documentum internals and tuning blog, Inside Documentum (`http://robineast.wordpress.com`), and contributed hundreds of posts to EMC support forums. These community efforts bore fruit in the form of the award of EMC MVP and acceptance into the EMC Elect program.

Toni Verbeiren graduated as a PhD in theoretical physics in 2003. He used to work on models of artificial neural networks, entailing mathematics, statistics, simulations, (lots of) data, and numerical computations. Since then, he has been active in the industry in diverse domains and roles: infrastructure management and deployment, service management, IT management, ICT/business alignment, and enterprise architecture. Around 2010, Toni started picking up his earlier passion, which was then named data science. The combination of data and common sense can be a very powerful basis to make decisions and analyze risk.

Toni is active as an owner and consultant at Data Intuitive (`http://www.data-intuitive.com/`) in everything related to big data science and its applications to decision and risk management. He is currently involved in Exascience Life Lab (`http://www.exascience.com/`) and the Visual Data Analysis Lab (`http://vda-lab.be/`), which is concerned with scaling up visual analysis of biological and chemical data.

I'd like to thank various employers, clients, and colleagues for the insight and wisdom they shared with me. I'm grateful to the Belgian and Flemish governments (FWO, IWT) for financial support of the aforementioned academic projects.

Lijie Xu is a PhD student at the Institute of Software, Chinese Academy of Sciences. His research interests focus on distributed systems and large-scale data analysis. He has both academic and industrial experience in Microsoft Research Asia, Alibaba Taobao, and Tencent. As an open source software enthusiast, he has contributed to Apache Spark and written a popular technical report, named *Spark Internals*, in Chinese at `https://github.com/JerryLead/SparkInternals/tree/master/markdown`.

Table of Contents

Preface

Apache Spark has captured the imagination of the analytics and big data developers, and rightfully so. In a nutshell, Spark enables distributed computing on a large scale in the lab or in production. Till now, the pipeline collect-store-transform was distinct from the Data Science pipeline reason-model, which was again distinct from the deployment of the analytics and machine learning models. Now, with Spark and technologies, such as Kafka, we can seamlessly span the data management and data science pipelines. We can build data science models on larger datasets, requiring not just sample data. However, whatever models we build can be deployed into production (with added work from engineering on the "ilities", of course). It is our hope that this book would enable an engineer to get familiar with the fundamentals of the Spark platform as well as provide hands-on experience on some of the advanced capabilities.

What this book covers

Chapter 1, *Installing Spark and Setting up your Cluster*, discusses some common methods for setting up Spark.

Chapter 2, *Using the Spark Shell*, introduces the command line for Spark. The Shell is good for trying out quick program snippets or just figuring out the syntax of a call interactively.

Chapter 3, *Building and Running a Spark Application*, covers Maven and sbt for compiling Spark applications.

Chapter 4, *Creating a SparkContext*, describes the programming aspects of the connection to a Spark server, for example, the SparkContext.

Chapter 5, *Loading and Saving Data in Spark*, deals with how we can get data in and out of a Spark environment.

Chapter 6, *Manipulating your RDD*, describes how to program the Resilient Distributed Datasets, which is the fundamental data abstraction in Spark that makes all the magic possible.

Chapter 7, *Spark SQL*, deals with the SQL interface in Spark. Spark SQL probably is the most widely used feature.

Chapter 8, *Spark with Big Data*, describes the interfaces with Parquet and HBase.

Chapter 9, *Machine Learning Using Spark MLlib*, talks about regression, classification, clustering, and recommendation. This is probably the largest chapter in this book. If you are stranded on a remote island and could take only one chapter with you, this should be the one!

Chapter 10, *Testing*, talks about the importance of testing distributed applications.

Chapter 11, Tips and Tricks, distills some of the things we have seen. Our hope is that as you get more and more adept in Spark programming, you will add this to the list and send us your gems for us to include in the next version of this book!

What you need for this book

Like any development platform, learning to develop systems with Spark takes trial and error. Writing programs, encountering errors, agonizing over pesky bugs are all part of the process. We expect a basic level of programming skills—Python or Java—and experience in working with operating system commands. We have kept the examples simple and to the point. In terms of resources, we do not assume any esoteric equipment for running the examples and developing the code. A normal development machine is enough.

Who this book is for

Data scientists and data engineers would benefit more from this book. Folks who have an exposure to big data and analytics will recognize the patterns and the pragmas. Having said that, anyone who wants to understand distributed programming would benefit from working through the examples and reading the book.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "While the methods for loading an RDD are largely found in the `SparkContext` class, the methods for saving an RDD are defined on the RDD classes."

A block of code is set as follows:

```
//Next two lines only needed if you decide to use the assembly plugin
import AssemblyKeys._assemblySettings

scalaVersion := "2.10.4"

name := "groupbytest"

libraryDependencies ++= Seq(
   "org.spark-project" % "spark-core_2.10" % "1.1.0"
)
```

Any command-line input or output is written as follows:

```
scala> val inFile = sc.textFile("./spam.data")
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: " Select **Source Code** from option **2. Choose a package type** and either download directly or select a mirror."

[Warnings or important notes appear in a box like this.]

[Tips and tricks appear like this.]

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1 Installing Spark and Setting up your Cluster

This chapter will detail some common methods to set up Spark. Spark on a single machine is excellent for testing or exploring small datasets, but here you will also learn to use Spark's built-in deployment scripts with a dedicated cluster via SSH (Secure Shell). This chapter will explain the use of Mesos and Hadoop clusters with YARN or Chef to deploy Spark. For Cloud deployments of Spark, this chapter will look at EC2 (both traditional and EC2MR). Feel free to skip this chapter if you already have your local Spark instance installed and want to get straight to programming.

Regardless of how you are going to deploy Spark, you will want to get the latest version of Spark from `https://spark.apache.org/downloads.html` (Version 1.2.0 as of this writing). Spark currently releases every 90 days. For coders who want to work with the latest builds, try cloning the code directly from the repository at `https://github.com/apache/spark`. The building instructions are available at `https://spark.apache.org/docs/latest/building-spark.html`. Both source code and prebuilt binaries are available at this link. To interact with **Hadoop Distributed File System** (**HDFS**), you need to use Spark, which is built against the same version of Hadoop as your cluster. For Version 1.1.0 of Spark, the prebuilt package is built against the available Hadoop Versions 1.x, 2.3, and 2.4. If you are up for the challenge, it's recommended that you build against the source as it gives you the flexibility of choosing which HDFS Version you want to support as well as apply patches with. In this chapter, we will do both.

To compile the Spark source, you will need the appropriate version of Scala and the matching JDK. The Spark source tar includes the required Scala components. The following discussion is only for information—there is no need to install Scala.

The Spark developers have done a good job of managing the dependencies. Refer to the `https://spark.apache.org/docs/latest/building-spark.html` web page for the latest information on this. According to the website, "Building Spark using Maven requires Maven 3.0.4 or newer and Java 6+." Scala gets pulled down as a dependency by Maven (currently Scala 2.10.4). Scala does not need to be installed separately, it is just a bundled dependency.

Just as a note, Spark 1.1.0 requires Scala 2.10.4 while the 1.2.0 version would run on 2.10 and Scala 2.11. I just saw e-mails in the Spark users' group on this.

[This brings up another interesting point about the Spark community. The two essential mailing lists are `user@spark.apache.org` and `dev@spark.apache.org`. More details about the Spark community are available at `https://spark.apache.org/community.html`.]

Directory organization and convention

One convention that would be handy is to download and install software in the `/opt` directory. Also have a generic soft link to Spark that points to the current version. For example, `/opt/spark` points to `/opt/spark-1.1.0` with the following command:

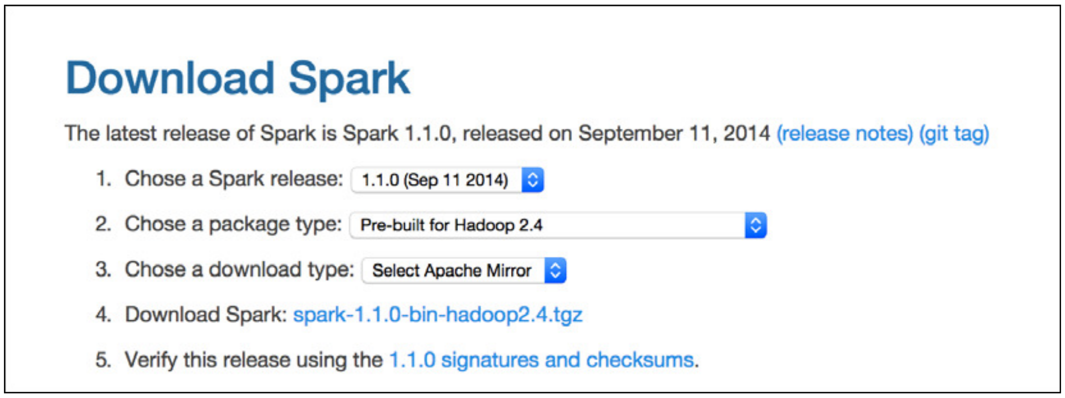
```
sudo ln -f -s spark-1.1.0 spark
```

Later, if you upgrade, say to Spark 1.2, you can change the softlink.

But remember to copy any configuration changes and old logs when you change to a new distribution. A more flexible way is to change the configuration directory to `/etc/opt/spark` and the log files to `/var/log/spark/`. That way, these will stay independent of the distribution updates. More details are available at `https://spark.apache.org/docs/latest/configuration.html#overriding-configuration-directory` and `https://spark.apache.org/docs/latest/configuration.html#configuring-logging`.

Installing prebuilt distribution

Let's download prebuilt Spark and install it. Later, we will also compile a Version and build from the source. The download is straightforward. The page to go to for this is `http://spark.apache.org/downloads.html`. Select the options as shown in the following screenshot:



Download Spark

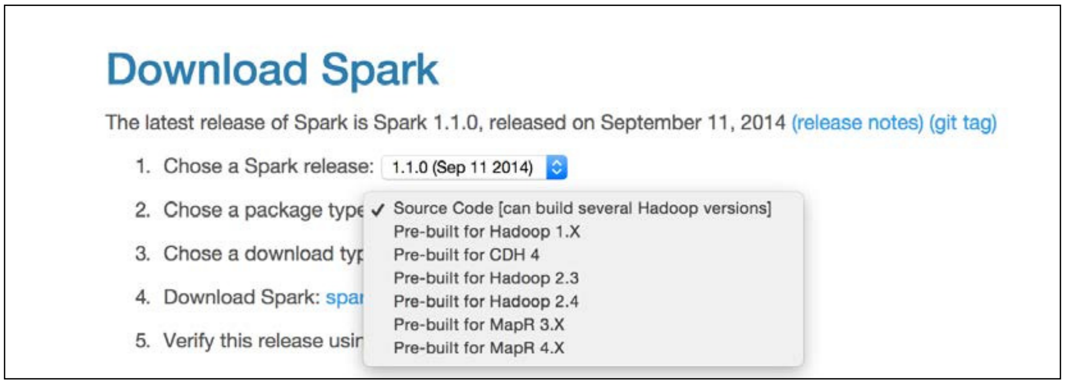
The latest release of Spark is Spark 1.1.0, released on September 11, 2014 (release notes) (git tag)

1. Chose a Spark release: 1.1.0 (Sep 11 2014)
2. Chose a package type: Pre-built for Hadoop 2.4
3. Chose a download type: Select Apache Mirror
4. Download Spark: spark-1.1.0-bin-hadoop2.4.tgz
5. Verify this release using the 1.1.0 signatures and checksums.

We will do a wget from the command line. You can do a direct download as well:

```
cd /opt
sudo wget http://apache.arvixe.com/spark/spark-1.1.1/spark-1.1.1-bin-
hadoop2.4.tgz
```

We are downloading the prebuilt version for Apache Hadoop 2.4 from one of the possible mirrors. We could have easily downloaded other prebuilt versions as well, as shown in the following screenshot:



Download Spark

The latest release of Spark is Spark 1.1.0, released on September 11, 2014 (release notes) (git tag)

1. Chose a Spark release: 1.1.0 (Sep 11 2014)
2. Chose a package type
3. Chose a download typ
4. Download Spark: spar
5. Verify this release usir

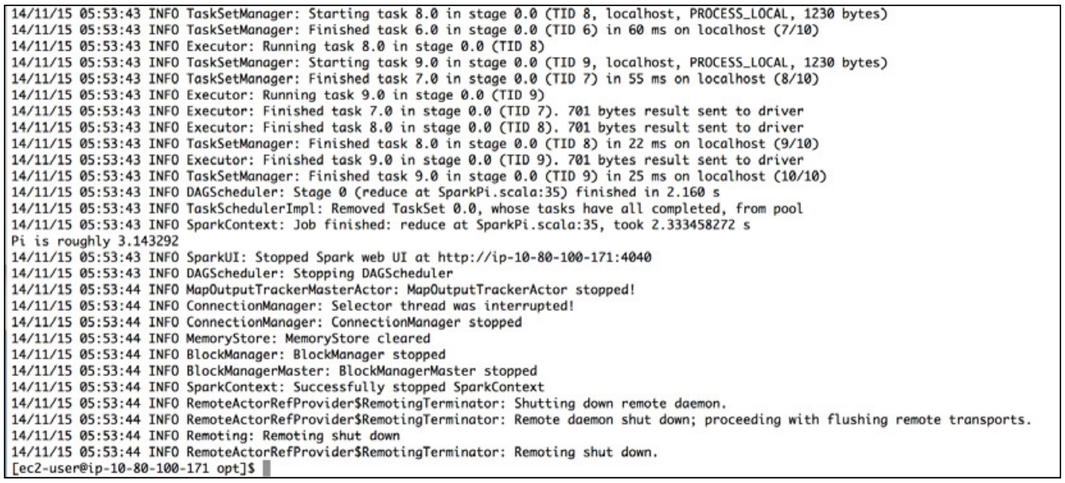
To uncompress it, execute the following command:

```
tar xvf spark-1.1.1-bin-hadoop2.4.tgz
```

To test the installation, run the following command:

```
/opt/spark-1.1.1-bin-hadoop2.4/bin/run-example SparkPi 10
```

It will fire up the Spark stack and calculate the value of Pi. The result should be as shown in the following screenshot:



```
14/11/15 05:53:43 INFO TaskSetManager: Starting task 8.0 in stage 0.0 (TID 8, localhost, PROCESS_LOCAL, 1230 bytes)
14/11/15 05:53:43 INFO TaskSetManager: Finished task 6.0 in stage 0.0 (TID 6) in 60 ms on localhost (7/10)
14/11/15 05:53:43 INFO Executor: Running task 8.0 in stage 0.0 (TID 8)
14/11/15 05:53:43 INFO TaskSetManager: Starting task 9.0 in stage 0.0 (TID 9, localhost, PROCESS_LOCAL, 1230 bytes)
14/11/15 05:53:43 INFO TaskSetManager: Finished task 7.0 in stage 0.0 (TID 7) in 55 ms on localhost (8/10)
14/11/15 05:53:43 INFO Executor: Running task 9.0 in stage 0.0 (TID 9)
14/11/15 05:53:43 INFO Executor: Finished task 7.0 in stage 0.0 (TID 7). 701 bytes result sent to driver
14/11/15 05:53:43 INFO Executor: Finished task 8.0 in stage 0.0 (TID 8). 701 bytes result sent to driver
14/11/15 05:53:43 INFO TaskSetManager: Finished task 8.0 in stage 0.0 (TID 8) in 22 ms on localhost (9/10)
14/11/15 05:53:43 INFO Executor: Finished task 9.0 in stage 0.0 (TID 9). 701 bytes result sent to driver
14/11/15 05:53:43 INFO TaskSetManager: Finished task 9.0 in stage 0.0 (TID 9) in 25 ms on localhost (10/10)
14/11/15 05:53:43 INFO DAGScheduler: Stage 0 (reduce at SparkPi.scala:35) finished in 2.160 s
14/11/15 05:53:43 INFO TaskSchedulerImpl: Removed TaskSet 0.0, whose tasks have all completed, from pool
14/11/15 05:53:43 INFO SparkContext: Job finished: reduce at SparkPi.scala:35, took 2.333458272 s
Pi is roughly 3.143292
14/11/15 05:53:43 INFO SparkUI: Stopped Spark web UI at http://ip-10-80-100-171:4040
14/11/15 05:53:43 INFO DAGScheduler: Stopping DAGScheduler
14/11/15 05:53:44 INFO MapOutputTrackerMasterActor: MapOutputTrackerActor stopped!
14/11/15 05:53:44 INFO ConnectionManager: Selector thread was interrupted!
14/11/15 05:53:44 INFO ConnectionManager: ConnectionManager stopped
14/11/15 05:53:44 INFO MemoryStore: MemoryStore cleared
14/11/15 05:53:44 INFO BlockManager: BlockManager stopped
14/11/15 05:53:44 INFO BlockManagerMaster: BlockManagerMaster stopped
14/11/15 05:53:44 INFO SparkContext: Successfully stopped SparkContext
14/11/15 05:53:44 INFO RemoteActorRefProvider$RemotingTerminator: Shutting down remote daemon.
14/11/15 05:53:44 INFO RemoteActorRefProvider$RemotingTerminator: Remote daemon shut down; proceeding with flushing remote transports.
14/11/15 05:53:44 INFO Remoting: Remoting shut down
14/11/15 05:53:44 INFO RemoteActorRefProvider$RemotingTerminator: Remoting shut down.
[ec2-user@ip-10-80-100-171 opt]$
```

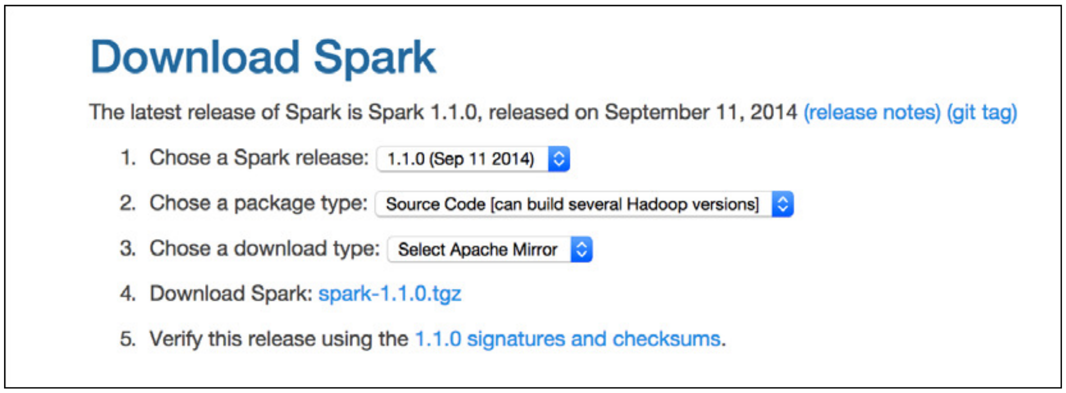
Building Spark from source

Let's compile Spark on a new AWS instance. That way you can clearly understand what all the requirements are to get a Spark stack compiled and installed. I am using the Amazon Linux AMI, which has Java and other base stack installed by default. As this is a book on Spark, we can safely assume that you would have the base configurations covered. We will cover the incremental installs for the Spark stack here.

The latest instructions for building from the source are available at `https://spark.apache.org/docs/latest/building-with-maven.html`.

Downloading the source

The first order of business is to download the latest source from `https://spark.apache.org/downloads.html`. Select **Source Code** from option **2. Chose a package type** and either download directly or select a mirror. The download page is shown in the following screenshot:



Download Spark

The latest release of Spark is Spark 1.1.0, released on September 11, 2014 (release notes) (git tag)

1. Chose a Spark release: 1.1.0 (Sep 11 2014)
2. Chose a package type: Source Code [can build several Hadoop versions]
3. Chose a download type: Select Apache Mirror
4. Download Spark: spark-1.1.0.tgz
5. Verify this release using the 1.1.0 signatures and checksums.

We can either download from the web page or use wget. We will do the wget from one of the mirrors, as shown in the following code:

```
cd /opt
sudo wget http://apache.arvixe.com/spark/spark-1.1.1/spark-1.1.1.tgz
sudo tar -xzf spark-1.1.1.tgz
```

The latest development source is in GitHub, which is available at `https://github.com/apache/spark`. The latest version can be checked out by the Git clone at `https://github.com/apache/spark.git`. This should be done only when you want to see the developments for the next version or when you are contributing to the source.

Compiling the source with Maven

Compilation by nature is uneventful, but a lot of information gets displayed on the screen:

```
cd /opt/spark-1.1.1
export MAVEN_OPTS="-Xmx2g -XX:MaxPermSize=512M
-XX:ReservedCodeCacheSize=512m"
mvn -Pyarn -Phadoop-2.4 -Dhadoop.version=2.4.0 -DskipTests clean
package
```

In order for the preceding snippet to work, we will need Maven installed in our system. In case Maven is not installed in your system, the commands to install the latest version of Maven are given here:

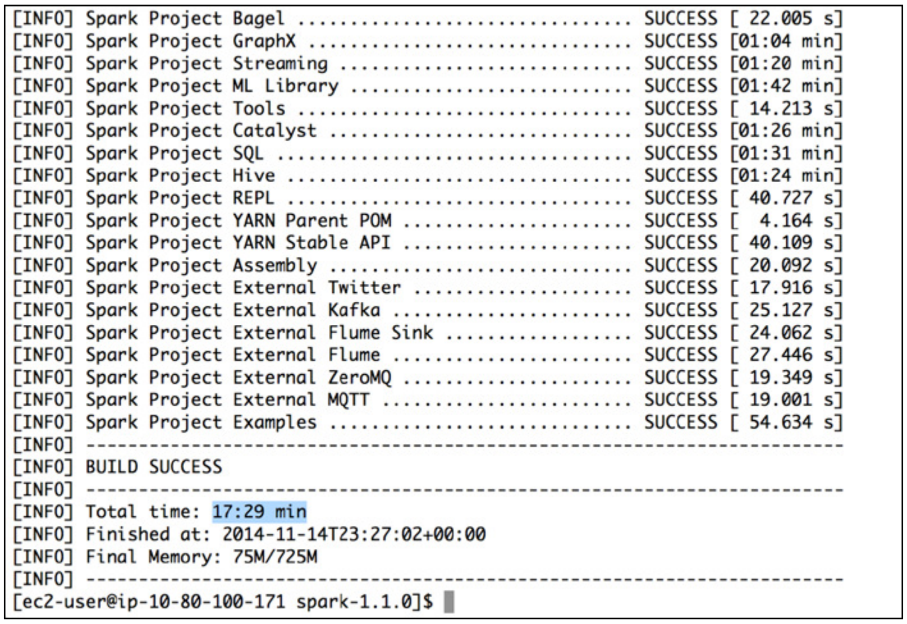
```
wget http://download.nextag.com/apache/maven/maven-
3/3.2.5/binaries/apache-maven-3.2.5-bin.tar.gz

sudo tar -xzf apache-maven-3.2.5-bin.tar.gz

sudo ln -f -s apache-maven-3.2.5 maven

export M2_HOME=/opt/maven

export PATH=${M2_HOME}/bin:${PATH}
```

Detailed Maven installation instructions are available at `http://maven.apache.org/download.cgi#Installation`.

Sometimes you will have to debug Maven using the –X switch. When I ran Maven, the Amazon Linux AMI didn't have the Java compiler! I had to install `javac` for Amazon Linux AMI using the following command:

```
sudo yum install java-1.7.0-openjdk-devel
```

The compilation time varies. On my Mac it took approximately 11 minutes. The Amazon Linux on a t2-medium instance took 18 minutes. In the end, you should see a build success message like the one shown in the following screenshot:



```
[INFO] Spark Project Bagel ................................ SUCCESS [ 22.005 s]
[INFO] Spark Project GraphX ............................... SUCCESS [01:04 min]
[INFO] Spark Project Streaming ............................ SUCCESS [01:20 min]
[INFO] Spark Project ML Library ........................... SUCCESS [01:42 min]
[INFO] Spark Project Tools ................................ SUCCESS [ 14.213 s]
[INFO] Spark Project Catalyst ............................. SUCCESS [01:26 min]
[INFO] Spark Project SQL .................................. SUCCESS [01:31 min]
[INFO] Spark Project Hive ................................. SUCCESS [01:24 min]
[INFO] Spark Project REPL ................................. SUCCESS [ 40.727 s]
[INFO] Spark Project YARN Parent POM ...................... SUCCESS [  4.164 s]
[INFO] Spark Project YARN Stable API ...................... SUCCESS [ 40.109 s]
[INFO] Spark Project Assembly ............................. SUCCESS [ 20.092 s]
[INFO] Spark Project External Twitter ..................... SUCCESS [ 17.916 s]
[INFO] Spark Project External Kafka ....................... SUCCESS [ 25.127 s]
[INFO] Spark Project External Flume Sink .................. SUCCESS [ 24.062 s]
[INFO] Spark Project External Flume ....................... SUCCESS [ 27.446 s]
[INFO] Spark Project External ZeroMQ ...................... SUCCESS [ 19.349 s]
[INFO] Spark Project External MQTT ........................ SUCCESS [ 19.001 s]
[INFO] Spark Project Examples ............................. SUCCESS [ 54.634 s]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 17:29 min
[INFO] Finished at: 2014-11-14T23:27:02+00:00
[INFO] Final Memory: 75M/725M
[INFO] ------------------------------------------------------------------------
[ec2-user@ip-10-80-100-171 spark-1.1.0]$
```

Compilation switches

As an example, the switches for compilation of `-Pyarn -Phadoop-2.4 -Dhadoop.version=2.4.0` are explained in `https://spark.apache.org/docs/latest/building-spark.html#specifying-the-hadoop-version`. `–D` defines a system property and `–P` defines a profile.

A typical compile configuration that I use (for YARN, Hadoop Version 2.6 with Hive support) is given here:

```
mvn clean package -Pyarn -Dyarn.version=2.6.0 -Phadoop-2.4 -
Dhadoop.version=2.6.0 -Phive -DskipTests
```

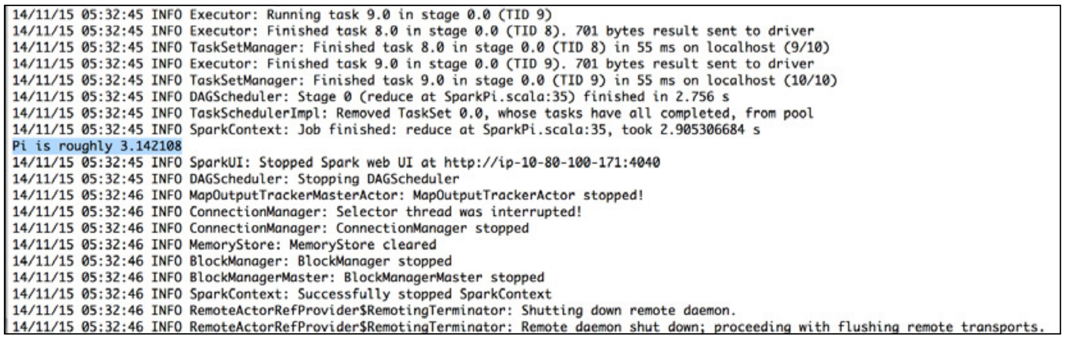
You can also compile the source code in IDEA and then upload the built Version to your cluster.

Testing the installation

A quick way to test the installation is by calculating Pi:

```
/opt/spark/bin/run-example SparkPi 10
```

The result should be a few debug messages and then the value of **Pi** as shown in the following screenshot:



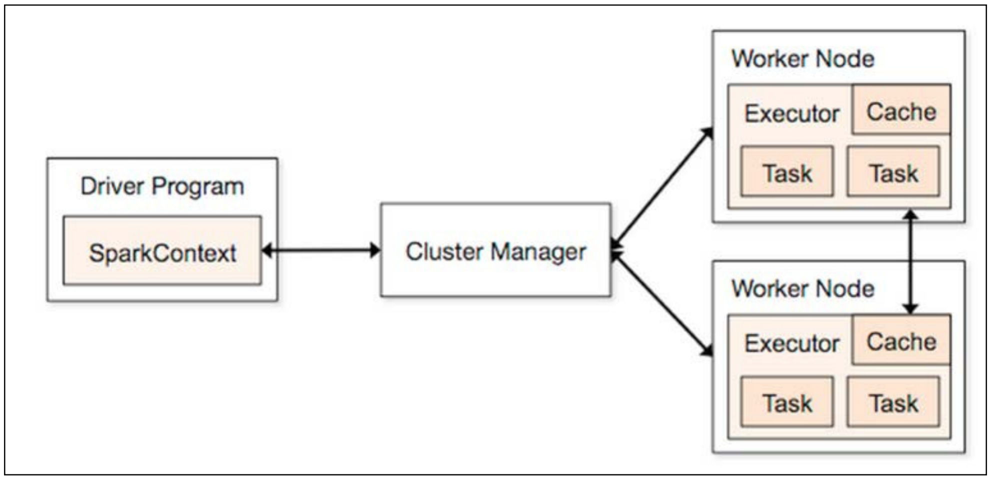
```
14/11/15 05:32:45 INFO Executor: Running task 9.0 in stage 0.0 (TID 9)
14/11/15 05:32:45 INFO Executor: Finished task 8.0 in stage 0.0 (TID 8). 701 bytes result sent to driver
14/11/15 05:32:45 INFO TaskSetManager: Finished task 8.0 in stage 0.0 (TID 8) in 55 ms on localhost (9/10)
14/11/15 05:32:45 INFO Executor: Finished task 9.0 in stage 0.0 (TID 9). 701 bytes result sent to driver
14/11/15 05:32:45 INFO TaskSetManager: Finished task 9.0 in stage 0.0 (TID 9) in 55 ms on localhost (10/10)
14/11/15 05:32:45 INFO DAGScheduler: Stage 0 (reduce at SparkPi.scala:35) finished in 2.756 s
14/11/15 05:32:45 INFO TaskSchedulerImpl: Removed TaskSet 0.0, whose tasks have all completed, from pool
14/11/15 05:32:45 INFO SparkContext: Job finished: reduce at SparkPi.scala:35, took 2.905306684 s
Pi is roughly 3.142108
14/11/15 05:32:45 INFO SparkUI: Stopped Spark web UI at http://ip-10-80-100-171:4040
14/11/15 05:32:45 INFO DAGScheduler: Stopping DAGScheduler
14/11/15 05:32:46 INFO MapOutputTrackerMasterActor: MapOutputTrackerActor stopped!
14/11/15 05:32:46 INFO ConnectionManager: Selector thread was interrupted!
14/11/15 05:32:46 INFO ConnectionManager: ConnectionManager stopped
14/11/15 05:32:46 INFO MemoryStore: MemoryStore cleared
14/11/15 05:32:46 INFO BlockManager: BlockManager stopped
14/11/15 05:32:46 INFO BlockManagerMaster: BlockManagerMaster stopped
14/11/15 05:32:46 INFO SparkContext: Successfully stopped SparkContext
14/11/15 05:32:46 INFO RemoteActorRefProvider$RemotingTerminator: Shutting down remote daemon.
14/11/15 05:32:46 INFO RemoteActorRefProvider$RemotingTerminator: Remote daemon shut down; proceeding with flushing remote transports.
```

Spark topology

This is a good time to talk about the basic mechanics and mechanisms of Spark. We will progressively dig deeper, but for now let's take a quick look at the top level.

Essentially, Spark provides a framework to process vast amounts of data, be it in gigabytes and terabytes and occasionally petabytes. The two main ingredients are computation and scale. The size and effectiveness of the problems we can solve depends on these two factors, that is, the ability to apply complex computations over large amounts of data in a timely fashion. If our monthly runs take 40 days, we have a problem. The key, of course, is parallelism, massive parallelism to be exact. We can make our computational algorithm tasks go parallel, that is instead of doing the steps one after another, we can perform many steps in parallel or carry out data parallelism, that is, we run the same algorithms over a partitioned dataset in parallel. In my humble opinion, Spark is extremely effective in data parallelism in an elegant framework.
As you will see in the rest of this book, the two components are **Resilient Distributed Dataset** (**RDD**) and cluster manager. The cluster manager distributes the code and manages the data that is represented in RDDs. RDDs with transformations and actions are the main programming abstractions and present parallelized collections. Behind the scenes, a cluster manager controls the distribution and interaction with RDDs, distributes code, and manages fault-tolerant execution. Spark works with three types of cluster managers – standalone, Apache Mesos, and Hadoop YARN. The Spark page at `http://spark.apache.org/docs/latest/cluster-overview.html` has a lot more details on this. I just gave you a quick introduction here.

If you have installed Hadoop 2.0, you are recommended to install Spark on YARN. If you have installed Hadoop 1.0, the standalone version is recommended. If you want to try Mesos, you can choose to install Spark on Mesos. Users are not recommended to install both YARN and Mesos.



The Spark driver program takes the program classes and hands them over to a cluster manager. The cluster manager, in turn, starts executors in multiple worker nodes, each having a set of tasks. When we ran the example program earlier, all these actions happened transparently in your machine! Later when we install in a cluster, the examples would run, again transparently, but across multiple machines in the cluster. That is the magic of Spark and distributed computing!

A single machine

A single machine is the simplest use case for Spark. It is also a great way to sanity check your build. In the `spark/bin` directory, there is a shell script called `run-example`, which can be used to launch a Spark job. The `run-example` script takes the name of a Spark class and some arguments. Earlier, we used the `run-example` script from the `/bin` directory to calculate the value of Pi. There is a collection of sample Spark jobs in `examples/src/main/scala/org/apache/spark/examples/`.

All of the sample programs take the parameter `master` (the cluster manager), which can be the URL of a distributed cluster or local[N], where N is the number of threads.

Going back to our `run-example` script, it invokes the more general `bin/spark-submit` script. For now, let's stick with the `run-example` script.

To run `GroupByTest` locally, try running the following code:

```
bin/run-example GroupByTest
```

It should produce an output like this given here:

```
14/11/15 06:28:40 INFO SparkContext: Job finished: count at
GroupByTest.scala:51, took 0.494519333 s

2000
```

Running Spark on EC2

The `ec2` directory contains the script to run a Spark cluster in EC2. These scripts can be used to run multiple Spark clusters and even run on spot instances. Spark can also be run on Elastic MapReduce, which is Amazon's solution for Map Reduce cluster management, and it gives you more flexibility around scaling instances. The Spark page at `http://spark.apache.org/docs/latest/ec2-scripts.html` has the latest on-running spark on EC2.

Running Spark on EC2 with the scripts

To get started, you should make sure you have EC2 enabled on your account by signing up at `https://portal.aws.amazon.com/gp/aws/manageYourAccount`. Then it is a good idea to generate a separate access key pair for your Spark cluster, which you can do at `https://portal.aws.amazon.com/gp/aws/securityCredentials`. You will also need to create an EC2 key pair so that the Spark script can SSH to the launched machines, which can be done at `https://console.aws.amazon.com/ec2/home` by selecting **Key Pairs** under **Network & Security**. Remember that key pairs are created per region, and so you need to make sure you create your key pair in the same region as you intend to run your Spark instances. Make sure to give it a name that you can remember as you will need it for the scripts (this chapter will use `spark-keypair` as its example key pair name.). You can also choose to upload your public SSH key instead of generating a new key. These are sensitive; so make sure that you keep them private. You also need to set `AWS_ACCESS_KEY` and `AWS_SECRET_KEY` as environment variables for the Amazon EC2 scripts:

```
chmod 400 spark-keypair.pem
export AWS_ACCESS_KEY_ID= AWSACcessKeyId
export AWS_SECRET_ACCESS_KEY=AWSSecretKey
```

You will find it useful to download the EC2 scripts provided by Amazon from `http://aws.amazon.com/developertools/Amazon-EC2/351`. Once you unzip the resulting zip file, you can add the bin to your `PATH` in a manner similar to what you did with the Spark bin:

```
wget http://s3.amazonaws.com/ec2-downloads/ec2-api-tools.zip
unzip ec2-api-tools.zip
cd ec2-api-tools-*
export EC2_HOME=`pwd`
export PATH=$PATH:`pwd`/bin
```

In order to test whether this works, try the following commands:

```
$ec2-describe-regions
```

This should display the following output:

```
REGION eu-central-1     ec2.eu-central-1.amazonaws.com
REGION sa-east-1        ec2.sa-east-1.amazonaws.com
REGION ap-northeast-1   ec2.ap-northeast-1.amazonaws.com
REGION eu-west-1        ec2.eu-west-1.amazonaws.com
REGION us-east-1        ec2.us-east-1.amazonaws.com
REGION us-west-1       ec2.us-west-1.amazonaws.com
REGION us-west-2       ec2.us-west-2.amazonaws.com
REGION ap-southeast-2  ec2.ap-southeast-2.amazonaws.com
REGION ap-southeast-1  ec2.ap-southeast-1.amazonaws.com
```

Finally, you can refer to the EC2 command line tools reference page `http://docs.aws.amazon.com/AWSEC2/latest/CommandLineReference/set-up-ec2-cli-linux.html` as it has all the gory details.

The Spark EC2 script automatically creates a separate security group and firewall rules for running the Spark cluster. By default, your Spark cluster will be universally accessible on port 8080, which is a somewhat poor form. Sadly, the `spark_ec2.py` script does not currently provide an easy way to restrict access to just your host. If you have a static IP address, I strongly recommend limiting access in `spark_ec2.py`; simply replace all instances of `0.0.0.0/0` with `[yourip]/32`. This will not affect intra-cluster communication as all machines within a security group can talk to each other by default.

Next, try to launch a cluster on EC2:

```
./ec2/spark-ec2 -k spark-keypair -i pk-[....].pem -s 1 launch
myfirstcluster
```

If you get an error message like `The requested Availability Zone is currently constrained and....`, you can specify a different zone by passing in the `--zone` flag.

The `-i` parameter (in the preceding command line) is provided for specifying the private key to log into the instance; `-i pk-[....].pem` represents the path to the private key.

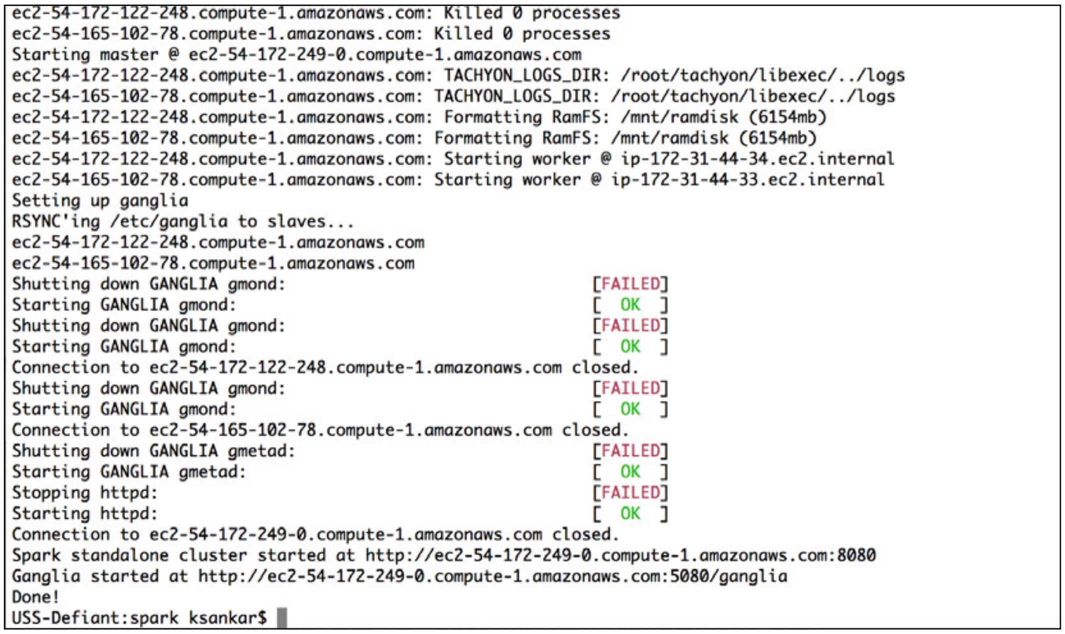
If you get an error about not being able to SSH to the master, make sure that only you have the permission to read the private key otherwise SSH will refuse to use it.

You may also encounter this error due to a race condition, when the hosts report themselves as alive but the Spark-`ec2` script cannot yet SSH to them. A fix for this issue is pending in `https://github.com/mesos/spark/pull/555`. For now, a temporary workaround until the fix is available in the version of Spark you are using is to simply sleep an extra 100 seconds at the start of `setup_cluster` using the `-w` parameter. The current script has 120 seconds of delay built in.

If you do get a transient error while launching a cluster, you can finish the launch process using the resume feature by running:

```
./ec2/spark-ec2 -i ~/spark-keypair.pem launch myfirstsparkcluster
--resume
```

It will go through a bunch of scripts, thus setting up Spark, Hadoop and so forth. If everything goes well, you should see something like the following screenshot:



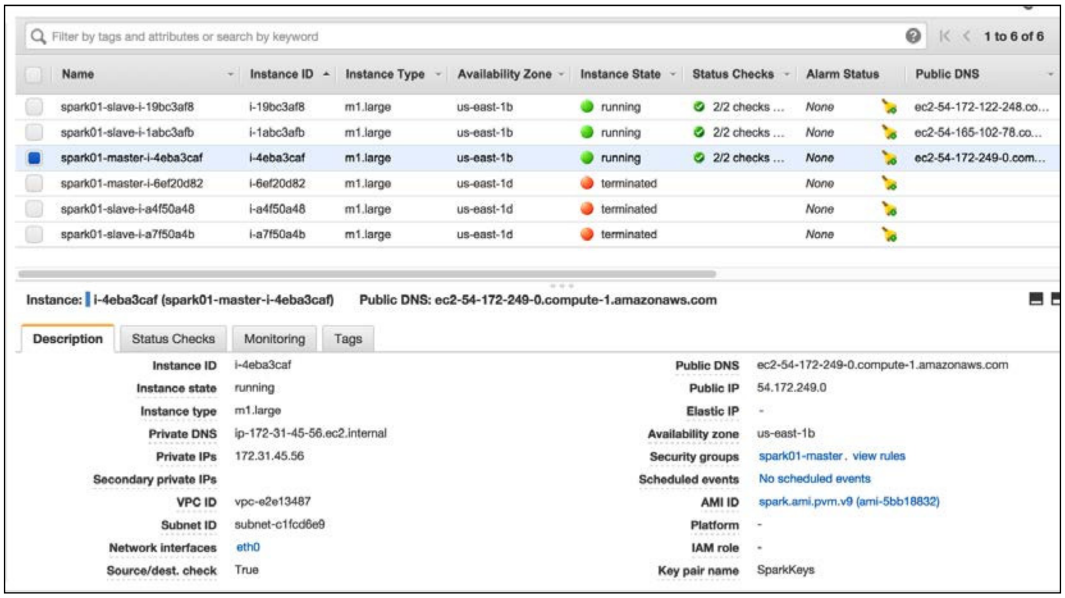
```
ec2-54-172-122-248.compute-1.amazonaws.com: Killed 0 processes
ec2-54-165-102-78.compute-1.amazonaws.com: Killed 0 processes
Starting master @ ec2-54-172-249-0.compute-1.amazonaws.com
ec2-54-172-122-248.compute-1.amazonaws.com: TACHYON_LOGS_DIR: /root/tachyon/libexec/../logs
ec2-54-165-102-78.compute-1.amazonaws.com: TACHYON_LOGS_DIR: /root/tachyon/libexec/../logs
ec2-54-172-122-248.compute-1.amazonaws.com: Formatting RamFS: /mnt/ramdisk (6154mb)
ec2-54-165-102-78.compute-1.amazonaws.com: Formatting RamFS: /mnt/ramdisk (6154mb)
ec2-54-172-122-248.compute-1.amazonaws.com: Starting worker @ ip-172-31-44-34.ec2.internal
ec2-54-165-102-78.compute-1.amazonaws.com: Starting worker @ ip-172-31-44-33.ec2.internal
Setting up ganglia
RSYNC'ing /etc/ganglia to slaves...
ec2-54-172-122-248.compute-1.amazonaws.com
ec2-54-165-102-78.compute-1.amazonaws.com
Shutting down GANGLIA gmond:                               [FAILED]
Starting GANGLIA gmond:                                    [  OK  ]
Shutting down GANGLIA gmond:                               [FAILED]
Starting GANGLIA gmond:                                    [  OK  ]
Connection to ec2-54-172-122-248.compute-1.amazonaws.com closed.
Shutting down GANGLIA gmond:                               [FAILED]
Starting GANGLIA gmond:                                    [  OK  ]
Connection to ec2-54-165-102-78.compute-1.amazonaws.com closed.
Shutting down GANGLIA gmetad:                              [FAILED]
Starting GANGLIA gmetad:                                   [  OK  ]
Stopping httpd:                                            [FAILED]
Starting httpd:                                            [  OK  ]
Connection to ec2-54-172-249-0.compute-1.amazonaws.com closed.
Spark standalone cluster started at http://ec2-54-172-249-0.compute-1.amazonaws.com:8080
Ganglia started at http://ec2-54-172-249-0.compute-1.amazonaws.com:5080/ganglia
Done!
USS-Defiant:spark ksankar$
```

This will give you a bare bones cluster with one master and one worker with all of the defaults on the default machine instance size. Next, verify that it started up and your firewall rules were applied by going to the master on port 8080. You can see in the preceding screenshot that the UI for the master is the output at the end of the script with port at 8080 and ganglia at 5080.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Your AWS EC2 dashboard will show the instances as follows:



Filter by tags and attributes or search by keyword — 1 to 6 of 6

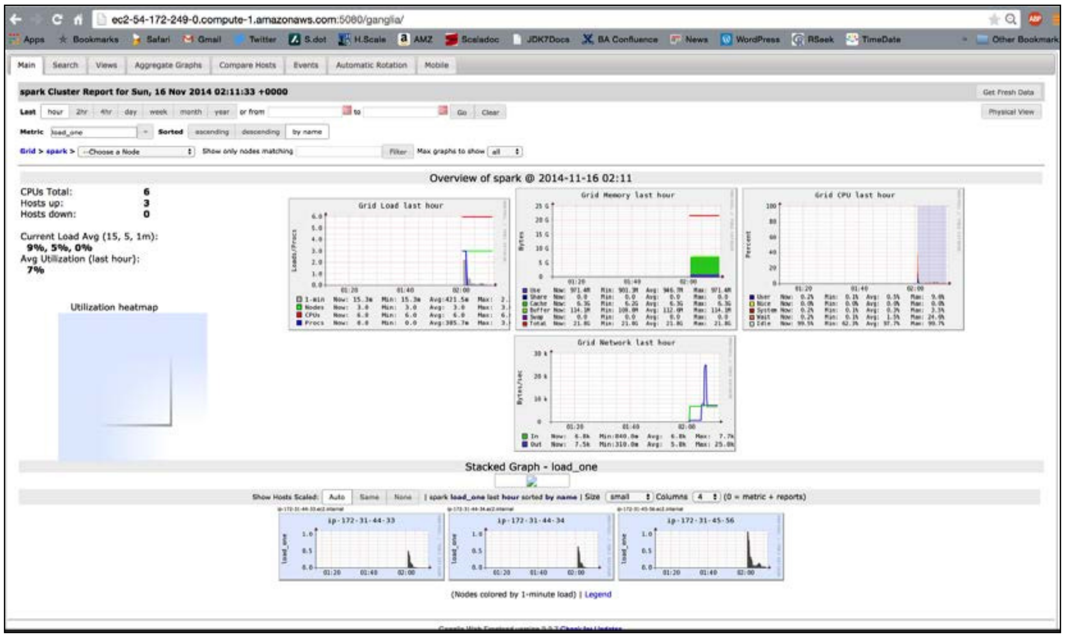
Name	Instance ID	Instance Type	Availability Zone	Instance State	Status Checks	Alarm Status	Public DNS
spark01-slave-i-19bc3af8	i-19bc3af8	m1.large	us-east-1b	running	2/2 checks ...	None	ec2-54-172-122-248.co...
spark01-slave-i-1abc3afb	i-1abc3afb	m1.large	us-east-1b	running	2/2 checks ...	None	ec2-54-165-102-78.co...
spark01-master-i-4eba3caf	i-4eba3caf	m1.large	us-east-1b	running	2/2 checks ...	None	ec2-54-172-249-0.com...
spark01-master-i-6ef20d82	i-6ef20d82	m1.large	us-east-1d	terminated		None	
spark01-slave-i-a4f50a48	i-a4f50a48	m1.large	us-east-1d	terminated		None	
spark01-slave-i-a7f50a4b	i-a7f50a4b	m1.large	us-east-1d	terminated		None	

Instance: i-4eba3caf (spark01-master-i-4eba3caf)   Public DNS: ec2-54-172-249-0.compute-1.amazonaws.com

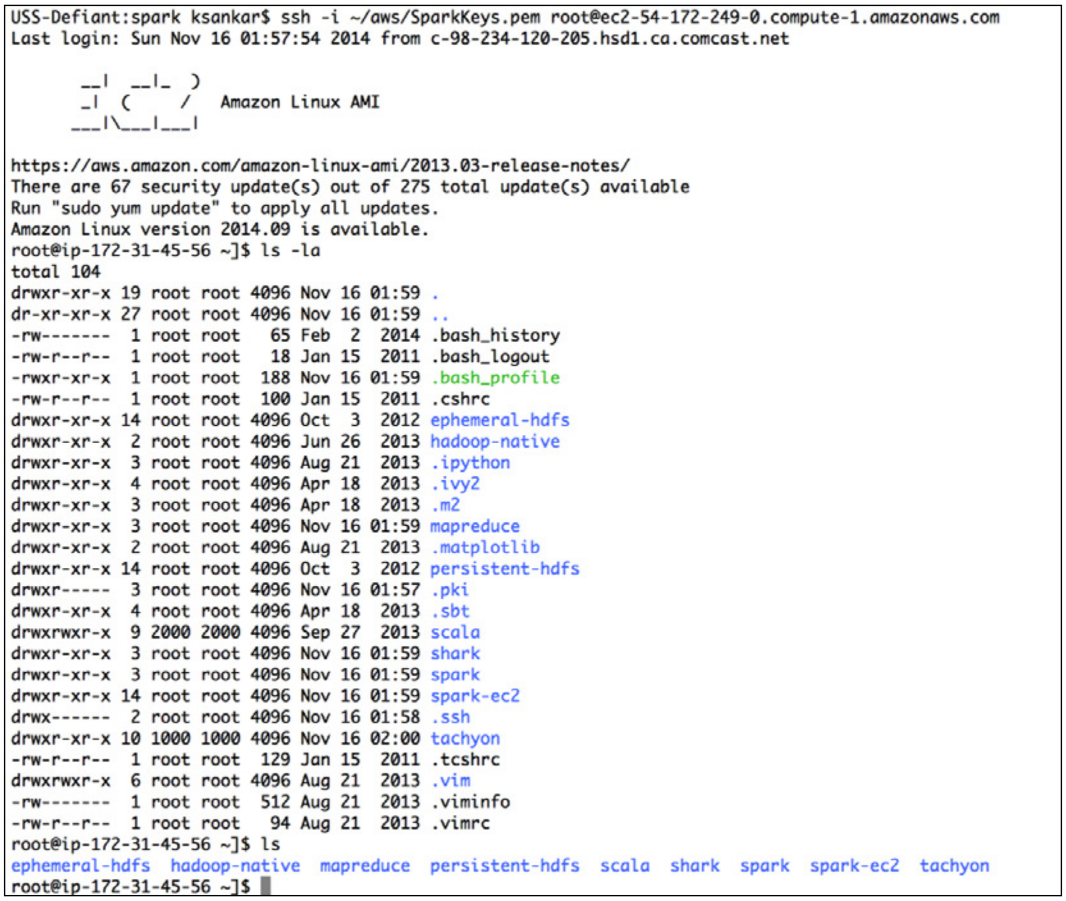
Description | Status Checks | Monitoring | Tags

Instance ID	i-4eba3caf	Public DNS	ec2-54-172-249-0.compute-1.amazonaws.com
Instance state	running	Public IP	54.172.249.0
Instance type	m1.large	Elastic IP	-
Private DNS	ip-172-31-45-56.ec2.internal	Availability zone	us-east-1b
Private IPs	172.31.45.56	Security groups	spark01-master, view rules
Secondary private IPs		Scheduled events	No scheduled events
VPC ID	vpc-e2e13487	AMI ID	spark.ami.pvm.v9 (ami-5bb18832)
Subnet ID	subnet-c1fcd6e9	Platform	-
Network interfaces	eth0	IAM role	-
Source/dest. check	True	Key pair name	SparkKeys

The ganglia dashboard shown in the following screenshot is a good place to monitor the instances:


ec2-54-172-249-0.compute-1.amazonaws.com:5080/ganglia/
spark Cluster Report for Sun, 16 Nov 2014 02:11:33 +0000
Overview of spark @ 2014-11-16 02:11
CPUs Total: 6
Hosts up: 3
Hosts down: 0
Current Load Avg (15, 5, 1m): 9%, 5%, 0%
Avg Utilization (last hour): 7%
Utilization heatmap
Grid Load last hour
Grid Memory last hour
Grid CPU last hour
Grid Network last hour
Stacked Graph - load_one
ip-172-31-44-33
ip-172-31-44-34
ip-172-31-45-56
(Nodes colored by 1-minute load) | Legend

Try running one of the example jobs on your new cluster to make sure everything is okay, as shown in the following screenshot:



```
USS-Defiant:spark ksankar$ ssh -i ~/aws/SparkKeys.pem root@ec2-54-172-249-0.compute-1.amazonaws.com
Last login: Sun Nov 16 01:57:54 2014 from c-98-234-120-205.hsd1.ca.comcast.net

       __|  __|_  )
       _|  (     /   Amazon Linux AMI
      ___|\___|___|

https://aws.amazon.com/amazon-linux-ami/2013.03-release-notes/
There are 67 security update(s) out of 275 total update(s) available
Run "sudo yum update" to apply all updates.
Amazon Linux version 2014.09 is available.
root@ip-172-31-45-56 ~]$ ls -la
total 104
drwxr-xr-x 19 root root 4096 Nov 16 01:59 .
dr-xr-xr-x 27 root root 4096 Nov 16 01:59 ..
-rw-------  1 root root   65 Feb  2  2014 .bash_history
-rw-r--r--  1 root root   18 Jan 15  2011 .bash_logout
-rwxr-xr-x  1 root root  188 Nov 16 01:59 .bash_profile
-rw-r--r--  1 root root  100 Jan 15  2011 .cshrc
drwxr-xr-x 14 root root 4096 Oct  3  2012 ephemeral-hdfs
drwxr-xr-x  2 root root 4096 Jun 26  2013 hadoop-native
drwxr-xr-x  3 root root 4096 Aug 21  2013 .ipython
drwxr-xr-x  4 root root 4096 Apr 18  2013 .ivy2
drwxr-xr-x  3 root root 4096 Apr 18  2013 .m2
drwxr-xr-x  3 root root 4096 Nov 16 01:59 mapreduce
drwxr-xr-x  2 root root 4096 Aug 21  2013 .matplotlib
drwxr-xr-x 14 root root 4096 Oct  3  2012 persistent-hdfs
drwxr-----  3 root root 4096 Nov 16 01:57 .pki
drwxr-xr-x  4 root root 4096 Apr 18  2013 .sbt
drwxrwxr-x  9 2000 2000 4096 Sep 27  2013 scala
drwxr-xr-x  3 root root 4096 Nov 16 01:59 shark
drwxr-xr-x  3 root root 4096 Nov 16 01:59 spark
drwxr-xr-x 14 root root 4096 Nov 16 01:59 spark-ec2
drwx------  2 root root 4096 Nov 16 01:58 .ssh
drwxr-xr-x 10 1000 1000 4096 Nov 16 02:00 tachyon
-rw-r--r--  1 root root  129 Jan 15  2011 .tcshrc
drwxrwxr-x  6 root root 4096 Aug 21  2013 .vim
-rw-------  1 root root  512 Aug 21  2013 .viminfo
-rw-r--r--  1 root root   94 Aug 21  2013 .vimrc
root@ip-172-31-45-56 ~]$ ls
ephemeral-hdfs  hadoop-native  mapreduce  persistent-hdfs  scala  shark  spark  spark-ec2  tachyon
root@ip-172-31-45-56 ~]$
```

The JPS should show this:

```
root@ip-172-31-45-56 ~]$ jps
1904 NameNode
2856 Jps
2426 Master
2078 SecondaryNameNode
```

The script has started Spark master, the Hadoop name node, and data nodes (in slaves).

Let's run the two programs that we ran earlier on our local machine:

```
cd spark
bin/run-example GroupByTest
bin/run-example SparkPi 10
```

The ease with which one can spin up a few nodes in the Cloud, install the Spark stack, and run the program in a distributed manner is interesting.

The `ec2/spark-ec2 destroy <cluster name>` command will terminate the instances.

Now that you've run a simple job on our EC2 cluster, it's time to configure your EC2 cluster for our Spark jobs. There are a number of options you can use to configure with the `spark-ec2` script.

The `ec2/ spark-ec2 –help` command will display all the options available.

First, consider what instance types you may need. EC2 offers an ever-growing collection of instance types and you can choose a different instance type for the master and the workers. The instance type has the most obvious impact on the performance of your Spark cluster. If your work needs a lot of RAM, you should choose an instance with more RAM. You can specify the instance type with `--instance-type=` (name of instance type). By default, the same instance type will be used for both the master and the workers; this can be wasteful if your computations are particularly intensive and the master isn't being heavily utilized. You can specify a different master instance type with `--master-instance-type=` (name of instance).

EC2 also has GPU instance types, which can be useful for workers but would be completely wasted on the master. This text will cover working with Spark and GPUs later on; however, it is important to note that EC2 GPU performance may be lower than what you get while testing locally due to the higher I/O overhead imposed by the hypervisor.

Spark's EC2 scripts use **Amazon Machine Images** (**AMI**) provided by the Spark team. Usually, they are current and sufficient for most of the applications. You might need your own AMI in case of circumstances like custom patches (for example, using a different version of HDFS) for Spark, as they will not be included in the machine image.

Deploying Spark on Elastic MapReduce

In addition to the Amazon basic EC2 machine offering, Amazon offers a hosted Map Reduce solution called **Elastic MapReduce** (**EMR**). Amazon provides a bootstrap script that simplifies the process of getting started using Spark on EMR. You will need to install the EMR tools from Amazon:

```
mkdir emr
cd emr
wget http://elasticmapreduce.s3.amazonaws.com/elastic-mapreduce-ruby.zip
unzip *.zip
```

This way the EMR scripts can access your AWS account you will want, to create a `credentials.json` file:

```
{
   "access-id": "<Your AWS access id here>",
   "private-key": "<Your AWS secret access key here>",
   "key-pair": "<The name of your ec2 key-pair here>",
   "key-pair-file": "<path to the .pem file for your ec2 key pair
   here>",
   "region": "<The region where you wish to launch your job flows
   (e.g us-east-1)>"
 }
```

Once you have the EMR tools installed, you can launch a Spark cluster by running:

```
elastic-mapreduce --create --alive --name "Spark/Shark Cluster" \
--bootstrap-action s3://elasticmapreduce/samples/spark/install-spark-
shark.sh \
--bootstrap-name "install Mesos/Spark/Shark" \
--ami-version 2.0  \
--instance-type m1.large --instance-count 2
```

This will give you a running EC2MR instance after about 5 to 10 minutes. You can list the status of the cluster by running `elastic-mapreduce -listode`. Once it outputs `j-[jobid]`, it is ready.

Deploying Spark with Chef (Opscode)

Chef is an open source automation platform that has become increasingly popular for deploying and managing both small and large clusters of machines. Chef can be used to control a traditional static fleet of machines and can also be used with EC2 and other cloud providers. Chef uses cookbooks as the basic building blocks of configuration and can either be generic or site-specific. If you have not used Chef before, a good tutorial for getting started with Chef can be found at `https://learnchef.opscode.com/`. You can use a generic Spark cookbook as the basis for setting up your cluster.

To get Spark working, you need to create a role for both the master and the workers as well as configure the workers to connect to the master. Start by getting the cookbook from `https://github.com/holdenk/chef-cookbook-spark`. The bare minimum need is setting the master hostname (as master) to enable worker nodes to connect and the username, so that Chef can be installed in the correct place. You will also need to either accept Sun's Java license or switch to an alternative JDK. Most of the settings that are available in `spark-env.sh` are also exposed through the cookbook settings. You can see an explanation of the settings in your section on "configuring multiple hosts over SSH". The settings can be set as per-role or you can modify the global defaults.

Create a role for the master with a knife role; create `spark_master_role -e [editor]`. This will bring up a template role file that you can edit. For a simple master, set it to this:

```
{
  "name": "spark_master_role",
  "description": "",
  "json_class": "Chef::Role",
  "default_attributes": {
  },
  "override_attributes": {
   "username":"spark",
   "group":"spark",
   "home":"/home/spark/sparkhome",
   "master_ip":"10.0.2.15",
  },
  "chef_type": "role",
  "run_list": [
   "recipe[spark::server]",
   "recipe[chef-client]",
  ],
  "env_run_lists": {
  }

}
```

Then create a role for the client in the same manner except that instead of `spark::server`, you need to use the `spark::client` recipe. Deploy the roles to different hosts:

```
knife node run_list add master role[spark_master_role]
knife node run_list add worker role[spark_worker_role]
```

Then run `chef-client` on your nodes to update. Congrats, you now have a Spark cluster running!

Deploying Spark on Mesos

Mesos is a cluster management platform for running multiple distributed applications or frameworks on a cluster. Mesos can intelligently schedule and run Spark, Hadoop, and other frameworks concurrently on the same cluster. Spark can be run on Mesos either by scheduling individual jobs as separate Mesos tasks or running all of Spark as a single Mesos task. Mesos can quickly scale up to handle large clusters beyond the size of which you would want to manage with plain old SSH scripts. Mesos, written in C++, was originally created at UC Berkley as a research project; it is currently undergoing Apache incubation and is actively used by Twitter.

The Spark web page has detailed instructions on installing and running Spark on Mesos.

To get started with Mesos, you can download the latest version from `http://mesos.apache.org/downloads/` and unpack it. Mesos has a number of different configuration scripts you can use; for an Ubuntu installation use `configure.ubuntu-lucid-64` and for other cases, the Mesos `README` file will point you at the configuration file you need to use. In addition to the requirements of Spark, you will need to ensure that you have the Python C header files installed (`python-dev` on Debian systems) or pass `--disable-python` to the configure script. Since Mesos needs to be installed on all the machines, you may find it easier to configure Mesos to install somewhere other than on the root, most easily alongside your Spark installation:

```
./configure --prefix=/home/sparkuser/mesos && make && make check &&
make install
```

Much like the configuration of Spark in standalone mode, with Mesos you need to make sure the different Mesos nodes can find each other. Start by having `mesossprefix/var/mesos/deploy/masters` to the hostname of the master and adding each worker hostname to `mesossprefix/var/mesos/deploy/slaves`. Then you will want to point the workers at the master (and possibly set some other values) in `mesossprefix/var/mesos/conf/mesos.conf`.

Once you have Mesos built, it's time to configure Spark to work with Mesos. This is as simple as copying the `conf/spark-env.sh.template` to `conf/spark-env.sh` and updating `MESOS_NATIVE_LIBRARY` to point to the path where Mesos is installed. You can find more information about the different settings in `spark-env.sh` in first table of the next section.

You will need to install both Mesos and Spark on all of the machines in your cluster. Once both Mesos and Spark are configured, you can copy the build to all of the machines using `pscp`, as shown in the following command:

```
pscp -v -r -h  -l sparkuser ./mesos /home/sparkuser/mesos
```

You can then start your Mesos clusters using `mesosprefix/sbin/mesos-start-cluster.sh` and schedule your Spark on Mesos by using `mesos://[host]:5050` as the master.

Spark on YARN

YARN is Apache Hadoop's NextGen MapReduce. The Spark project provides an easy way to schedule jobs on YARN once you have a Spark assembly built. The Spark web page `http://spark.apache.org/docs/latest/running-on-yarn.html` has the configuration details for YARN, which we had built earlier for when compiling with the –Pyarn switch. It is important that the Spark job you create uses a standalone master URL. The example Spark applications all read the master URL from the command line arguments; so specify `--args standalone`.

To run the same example as given in the SSH section, write the following commands:

```
sbt/sbt assembly #Build the assembly

SPARK_JAR=./core/target/spark-core-assembly-1.1.0.jar ./run
spark.deploy.yarn.Client --jar examples/target/scala-2.9.2/spark-
examples_2.9.2-0.7.0.jar --class spark.examples.GroupByTest --args
standalone --num-workers 2 --worker-memory 1g --worker-cores 1
```

Spark Standalone mode

If you have a set of machines without any existing cluster management software, you can deploy Spark over SSH with some handy scripts. This method is known as "**standalone mode**" in the Spark documentation at `http://spark.apache.org/docs/latest/spark-standalone.html`. An individual master and worker can be started by `sbin/start-master.sh` and `sbin/start-slaves.sh` respectively. The default port for the master is 8080. As you likely don't want to go to each of your machines and run these commands by hand, there are a number of helper scripts in `bin/` to help you run your servers.

A prerequisite for using any of the scripts is having password-less SSH access set up from the master to all of the worker machines. You probably want to create a new user for running Spark on the machines and lock it down. This book uses the username "sparkuser". On your master, you can run `ssh-keygen` to generate the SSH keys and make sure that you do not set a password. Once you have generated the key, add the public one (if you generated an RSA key, it would be stored in `~/.ssh/id_rsa.pub` by default) to `~/.ssh/authorized_keys2` on each of the hosts.

The Spark administration scripts require that your usernames match. If this isn't the case, you can configure an alternative username in your `~/.ssh/config`.

Now that you have the SSH access to the machines set up, it is time to configure Spark. There is a simple template in `[filepath]conf/spark-env.sh.template[/filepath]`, which you should copy to `[filepath]conf/spark-env.sh[/filepath]`. You will need to set `SCALA_HOME` to the path where you extracted Scala to. You may also find it useful to set some (or all) of the following environment variables:

Name	Purpose	Default
`MESOS_NATIVE_LIBRARY`	Point to math where Mesos lives	None
`SCALA_HOME`	Point to where you extracted Scala	None, must be set
`SPARK_MASTER_IP`	The IP address for the master to listen on and the IP address for the workers to connect to.	The result of running hostname
`SPARK_MASTER_PORT`	The port # for the Spark master to listen on	7077
`SPARK_MASTER_WEBUI_PORT`	The port # of the WEB UI on the master	8080
`SPARK_WORKER_CORES`	Number of cores to use	All of them
`SPARK_WORKER_MEMORY`	How much memory to use	Max of (system memory - 1GB, 512MB)
`SPARK_WORKER_PORT`	What port # the worker runs on	Rand
`SPARK_WEBUI_PORT`	What port # the worker WEB UI runs on	8081
`SPARK_WORKER_DIR`	Where to store files from the worker	`SPARK_HOME/work_dir`

Once you have your configuration done, it's time to get your cluster up and running. You will want to copy the version of Spark and the configuration you have built to all of your machines. You may find it useful to install `pssh`, a set of parallel SSH tools including `pscp`. The `pscp` makes it easy to scp to a number of target hosts, although it will take a while, as shown here:

```
pscp -v -r -h conf/slaves -l sparkuser ../opt/spark ~/
```

If you end up changing the configuration, you need to distribute the configuration to all of the workers, as shown here:

```
pscp -v -r -h conf/slaves -l sparkuser conf/spark-env.sh
/opt/spark/conf/spark-env.sh
```

> If you use a shared NFS on your cluster, while by default Spark names log files and similar with shared names, you should configure a separate worker directory, otherwise they will be configured to write to the same place. If you want to have your worker directories on the shared NFS, consider adding `` `hostname` `` for example `` SPARK_WORKER_DIR=~/work-`hostname` ``.
>
> You should also consider having your log files go to a scratch directory for performance.

Then you are ready to start the cluster and you can use the `sbin/start-all.sh`, `sbin/start-master.sh` and `sbin/start-slaves.sh` scripts. It is important to note that `start-all.sh` and `start-master.sh` both assume that they are being run on the node, which is the master for the cluster. The start scripts all daemonize, and so you don't have to worry about running them in a screen:

```
ssh master bin/start-all.sh
```

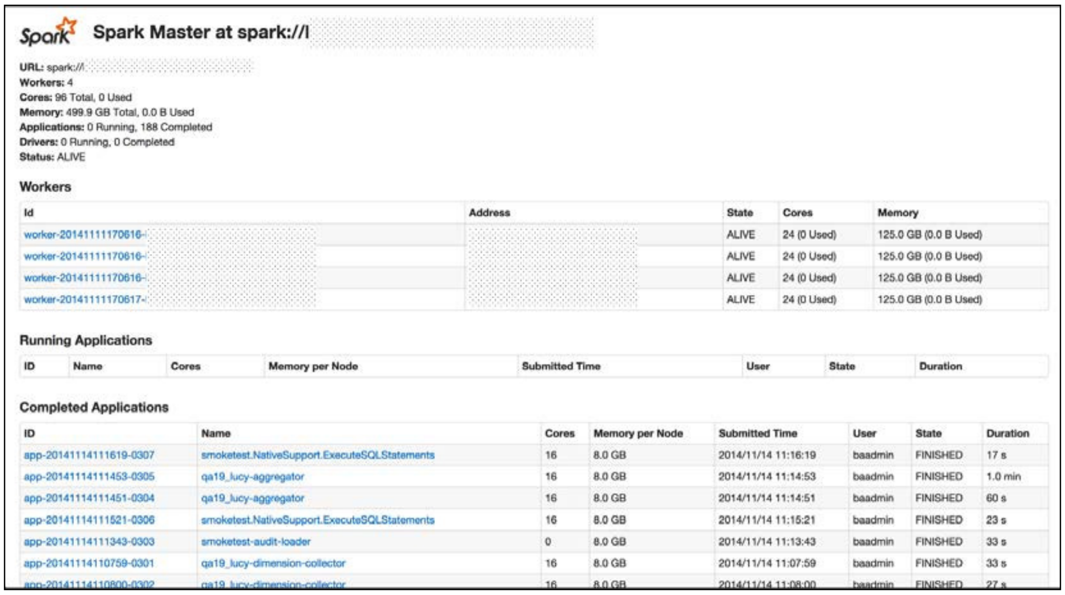
If you get a class not found error stating "`java.lang.NoClassDefFoundError: scala/ScalaObject`", check to make sure that you have Scala installed on that worker host and that the `SCALA_HOME` is set correctly.

> The Spark scripts assume that your master has Spark installed in the same directory as your workers. If this is not the case, you should edit `bin/spark-config.sh` and set it to the appropriate directories.

The commands provided by Spark to help you administer your cluster are given in the following table. More details are available in the Spark website at `http://spark.apache.org/docs/latest/spark-standalone.html#cluster-launch-scripts`.

Command	Use
`bin/slaves.sh <command>`	Runs the provided command on all of the worker hosts. For example, `bin/slave.sh` uptime will show how long each of the worker hosts have been up.
`bin/start-all.sh`	Starts the master and all of the worker hosts. Must be run on the master.
`bin/start-master.sh`	Starts the master host. Must be run on the master.
`bin/start-slaves.sh`	Starts the worker hosts.
`bin/start-slave.sh`	Start a specific worker.
`bin/stop-all.sh`	Stops master and workers.
`bin/stop-master.sh`	Stops the master.
`bin/stop-slaves.sh`	Stops all the workers.

You now have a running Spark cluster, as shown in the following screenshot! There is a handy Web UI on the master running on port 8080 you should go and visit, and on all of the workers on port 8081. The Web UI contains such helpful information as the current workers, and current and past jobs.



Spark Master at spark://I [redacted]

URL: spark://I [redacted]
Workers: 4
Cores: 96 Total, 0 Used
Memory: 499.9 GB Total, 0.0 B Used
Applications: 0 Running, 188 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers

Id	Address	State	Cores	Memory
worker-20141111170616-[redacted]	[redacted]	ALIVE	24 (0 Used)	125.0 GB (0.0 B Used)
worker-20141111170616-[redacted]	[redacted]	ALIVE	24 (0 Used)	125.0 GB (0.0 B Used)
worker-20141111170616-[redacted]	[redacted]	ALIVE	24 (0 Used)	125.0 GB (0.0 B Used)
worker-20141111170617-[redacted]	[redacted]	ALIVE	24 (0 Used)	125.0 GB (0.0 B Used)

Running Applications

ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration

Completed Applications

ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration
app-20141114111619-0307	smoketest.NativeSupport.ExecuteSQLStatements	16	8.0 GB	2014/11/14 11:16:19	baadmin	FINISHED	17 s
app-20141114111453-0305	qa19_lucy-aggregator	16	8.0 GB	2014/11/14 11:14:53	baadmin	FINISHED	1.0 min
app-20141114111451-0304	qa19_lucy-aggregator	16	8.0 GB	2014/11/14 11:14:51	baadmin	FINISHED	60 s
app-20141114111521-0306	smoketest.NativeSupport.ExecuteSQLStatements	16	8.0 GB	2014/11/14 11:15:21	baadmin	FINISHED	23 s
app-20141114111343-0303	smoketest-audit-loader	0	8.0 GB	2014/11/14 11:13:43	baadmin	FINISHED	33 s
app-20141114110759-0301	qa19_lucy-dimension-collector	16	8.0 GB	2014/11/14 11:07:59	baadmin	FINISHED	33 s
app-20141114110800-0302	qa19_lucy-dimension-collector	16	8.0 GB	2014/11/14 11:08:00	baadmin	FINISHED	27 s

Now that you have a cluster up and running, let's actually do something with it. As with the single host example, you can use the provided run script to run Spark commands. All of the examples listed in `examples/src/main/scala/spark/org/apache/spark/examples/` take a parameter, master, which points them to the master. Assuming that you are on the master host, you could run them like this:

```
./run-example GroupByTest spark://`hostname`:7077
```

If you run into an issue with `java.lang.UnsupportedClassVersionError`, you may need to update your JDK or recompile Spark if you grabbed the binary version. Version 1.1.0 was compiled with JDK 1.7 as the target. You can check the version of the JRE targeted by Spark with the following commands:

```
java -verbose -classpath ./core/target/scala-2.9.2/classes/
spark.SparkFiles |head -n 20
Version 49 is JDK1.5, Version 50 is JDK1.6 and Version 60 is JDK1.7
```

If you can't connect to localhost, make sure that you've configured your master (`spark.driver.port`) to listen to all of the IP addresses (or if you don't want to replace localhost with the IP address configured to listen to). More port configurations are listed at `http://spark.apache.org/docs/latest/configuration.html#networking`.

If everything has worked correctly, you will see the following log messages output to `stdout`:

```
13/03/28 06:35:31 INFO spark.SparkContext: Job finished: count at
GroupByTest.scala:35, took 2.482816756 s
2000
```

References:

- `http://archive09.linux.com/feature/151340`
- `http://spark-project.org/docs/latest/spark-standalone.html`
- `http://bickson.blogspot.com/2012/10/deploying-graphlabsparkmesos-cluster-on.html`
- `http://www.ibm.com/developerworks/library/os-spark/`
- `http://mesos.apache.org/`
- `http://aws.amazon.com/articles/Elastic-MapReduce/4926593393724923`
- `http://spark-project.org/docs/latest/ec2-scripts.html`
- `http://spark.apache.org/docs/latest/cluster-overview.html`

- `https://www.cs.berkeley.edu/~matei/papers/2012/nsdi_spark.pdf`
- `http://research.google.com/pubs/pub41378.html`
- `http://aws.amazon.com/articles/4926593393724923`
- `http://docs.aws.amazon.com/ElasticMapReduce/latest/DeveloperGuide/emr-cli-install.html`

Summary

In this chapter, we have gotten Spark installed on our machine for local development and set up on our cluster, and so we are ready to run the applications that we write. While installing and maintaining a cluster is a good option, Spark is also available as a service option from Databricks. Databricks' upcoming Databricks Cloud for Spark available at `http://databricks.com/product` is a very convenient offering for anyone who does not want to deal with the set up/maintenance of the cluster. They have the concept of a big data pipeline — from ETL to Analytics. This looks truly interesting to explore!

In the next chapter, you will learn to use the Spark shell.

2
Using the Spark Shell

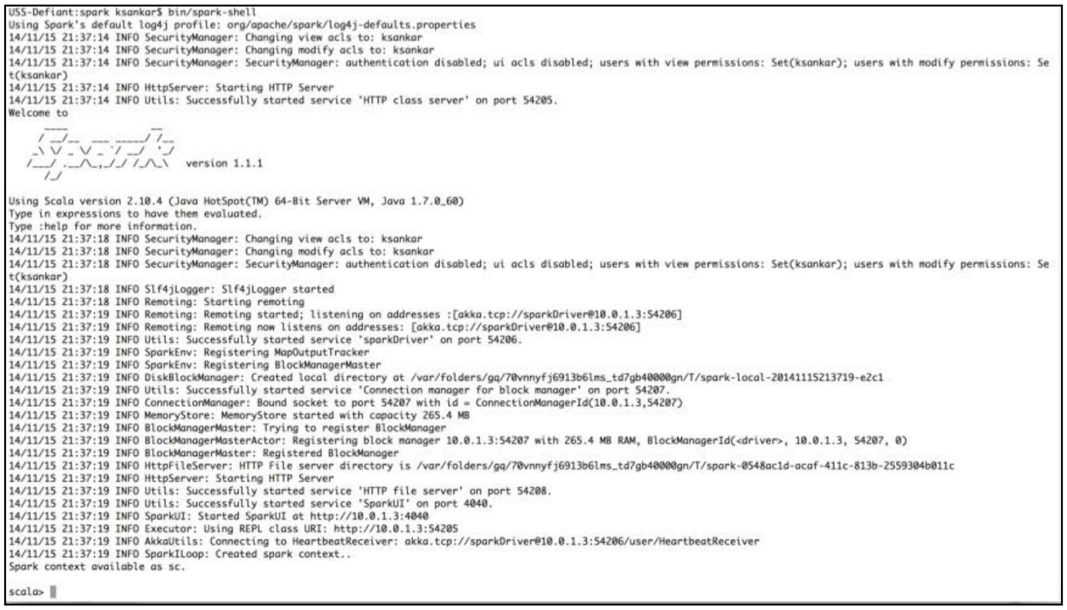
The Spark shell is a wonderful tool for rapid prototyping with Spark. It helps to be familiar with Scala, but that isn't necessary. The Spark shell works with Scala and Python. The Spark shell allows you to interactively query and communicate with the Spark cluster. This can be great for debugging, for just trying things out, or interactively exploring new datasets or approaches. The previous chapter should have gotten you to the point of having a Spark instance running, so now, all you need to do is start your Spark shell and point it at your running instance with the command given in the next few lines. Spark will start an instance when you invoke the Spark shell or start a Spark program from an IDE. So, a local installation on a Mac or Linux PC/laptop is sufficient to start exploring the Spark shell. Not having to spin up a real cluster to do the prototyping is an important feature of Spark.

Assuming that you have installed Spark in the `/opt` directory and have a soft link to Spark, run the following commands:

```
cd /opt/spark
export MASTER=spark://`hostname`:7077
bin/spark-shell
```

If you are running Spark in the local mode and don't have a Spark instance already running, you can just run the preceding command without the `MASTER=` part. As a result, the shell will run with only one thread; you can specify `local[n]` to run *n* threads.

You will see the shell prompt as shown in the following screenshot:



```
USS-Defiant:spark ksankar$ bin/spark-shell
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
14/11/15 21:37:14 INFO SecurityManager: Changing view acls to: ksankar
14/11/15 21:37:14 INFO SecurityManager: Changing modify acls to: ksankar
14/11/15 21:37:14 INFO SecurityManager: SecurityManager: authentication disabled; ui acls disabled; users with view permissions: Set(ksankar); users with modify permissions: Se
t(ksankar)
14/11/15 21:37:14 INFO HttpServer: Starting HTTP Server
14/11/15 21:37:14 INFO Utils: Successfully started service 'HTTP class server' on port 54205.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.1.1
      /_/

Using Scala version 2.10.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_60)
Type in expressions to have them evaluated.
Type :help for more information.
14/11/15 21:37:18 INFO SecurityManager: Changing view acls to: ksankar
14/11/15 21:37:18 INFO SecurityManager: Changing modify acls to: ksankar
14/11/15 21:37:18 INFO SecurityManager: SecurityManager: authentication disabled; ui acls disabled; users with view permissions: Set(ksankar); users with modify permissions: Se
t(ksankar)
14/11/15 21:37:18 INFO Slf4jLogger: Slf4jLogger started
14/11/15 21:37:18 INFO Remoting: Starting remoting
14/11/15 21:37:19 INFO Remoting: Remoting started; listening on addresses :[akka.tcp://sparkDriver@10.0.1.3:54206]
14/11/15 21:37:19 INFO Remoting: Remoting now listens on addresses: [akka.tcp://sparkDriver@10.0.1.3:54206]
14/11/15 21:37:19 INFO Utils: Successfully started service 'sparkDriver' on port 54206.
14/11/15 21:37:19 INFO SparkEnv: Registering MapOutputTracker
14/11/15 21:37:19 INFO SparkEnv: Registering BlockManagerMaster
14/11/15 21:37:19 INFO DiskBlockManager: Created local directory at /var/folders/gq/70vnnyfj6913b6lms_td7gb40000gn/T/spark-local-20141115213719-e2c1
14/11/15 21:37:19 INFO Utils: Successfully started service 'Connection manager for block manager' on port 54207.
14/11/15 21:37:19 INFO ConnectionManager: Bound socket to port 54207 with id = ConnectionManagerId(10.0.1.3,54207)
14/11/15 21:37:19 INFO MemoryStore: MemoryStore started with capacity 265.4 MB
14/11/15 21:37:19 INFO BlockManagerMaster: Trying to register BlockManager
14/11/15 21:37:19 INFO BlockManagerMasterActor: Registering block manager 10.0.1.3:54207 with 265.4 MB RAM, BlockManagerId(<driver>, 10.0.1.3, 54207, 0)
14/11/15 21:37:19 INFO BlockManagerMaster: Registered BlockManager
14/11/15 21:37:19 INFO HttpFileServer: HTTP File server directory is /var/folders/gq/70vnnyfj6913b6lms_td7gb40000gn/T/spark-0548ac1d-acaf-411c-813b-2559304b011c
14/11/15 21:37:19 INFO HttpServer: Starting HTTP Server
14/11/15 21:37:19 INFO Utils: Successfully started service 'HTTP file server' on port 54208.
14/11/15 21:37:19 INFO Utils: Successfully started service 'SparkUI' on port 4040.
14/11/15 21:37:19 INFO SparkUI: Started SparkUI at http://10.0.1.3:4040
14/11/15 21:37:19 INFO Executor: Using REPL class URI: http://10.0.1.3:54205
14/11/15 21:37:19 INFO AkkaUtils: Connecting to HeartbeatReceiver: akka.tcp://sparkDriver@10.0.1.3:54206/user/HeartbeatReceiver
14/11/15 21:37:19 INFO SparkILoop: Created spark context..
Spark context available as sc.

scala> 
```

Loading a simple text file

While running a Spark shell and connecting to an existing cluster, you should see something specifying the `app ID` such as "Connected to Spark cluster with `app ID` app-20130330015119-0001." The `app ID` will match the application entry as shown in the Web UI under running applications (by default, it will be viewable on port 4040). Start by downloading a dataset to use for some experimentation. There are a number of datasets put together for *The Elements of Statistical Learning*, which are in a very convenient form to use. Grab the spam dataset using the following command:

```
wget http://www-stat.stanford.edu/~tibs/ElemStatLearn/
datasets/spam.data
```

Alternatively, you can find the spam dataset from the GitHub link at `https://github.com/xsankar/fdps-vii`.

Now, load it as a text file into Spark with the following command inside your Spark shell:

```
scala> val inFile = sc.textFile("./spam.data")
```

This loads the `spam.data` file into Spark with each line being a separate entry in the **Resilient Distributed Datasets** (**RDD**). You will see RDDs in the later chapters, but RDD, in brief, is the basic data structure that Spark relies on. RDDs are very versatile in terms of scaling, computation capabilities, and transformations.

The path assumes that the data would be in the `/opt/spark` directory. Please type in the appropriate directory where you have downloaded the data.

The `sc` in the command line is the Spark context. While applications would create a Spark context explicitly, the Spark shell creates one called `sc` for you and that is the one we normally use.

Note: If you've connected to a Spark master, it's possible that it will attempt to load the file on any one of the different machines in the cluster, so make sure that it can be accessed by all the worker nodes in the cluster. In general you will want to put your data in HDFS, S3, or a similar distributed file systems for the future to avoid this problem. In a local mode, you can just load the file directly (for example, `sc.textFile([filepath])`). You can also use the `addFile` function on the Spark context to make a file available across all of the machines like this:

```
scala> import org.apache.spark.SparkFiles
scala> val file = sc.addFile("/opt/spark/spam.data")
scala> val inFile = sc.textFile(SparkFiles.get("spam.data"))
```

Just like most shells, the Spark shell has a command history; you can press the up arrow key to get to the previous commands. Are you getting tired of typing or not sure what method you want to call on an object? Press *Tab*, and the Spark shell will autocomplete the line of code in the best way it can.

For this example, the RDD with each line as an individual string isn't super useful as our input data is actually space separated numerical information. We can use the `map()` operation to iterate over the elements of the RDD and quickly convert it to a usable format (Note: `_.toDouble` is the Scala syntactic sugar for `x => x.toDouble`). We use one map operation to convert the line to a set of numbers in string format and then convert each of the number to a double, as shown next:

```
scala> val nums = inFile.map(line => line.split(' ').map(_.toDouble))
```

Verify that this is what we want by inspecting some elements in the `nums` RDD and comparing them against the original string RDD. Take a look at the first element of each by calling `.first()` on the RDDs:

> Most of the output following these commands is extraneous `INFO` messages. It is informative to see what Spark is doing under the covers. But if you want to keep the detailed messages out, you can copy `log4j.properties` into the current directory and set the `log4j.rootCategory` to `ERROR` instead of `INFO`. Then none of these messages will appear and it will be possible to concentrate just on the commands and the output.

```
scala> inFile.first()
[...]
14/11/15 23:46:41 INFO TaskSchedulerImpl: Removed TaskSet 0.0, whose
tasks have all completed, from pool
14/11/15 23:46:41 INFO DAGScheduler: Stage 0 (first at <console>:15)
finished in 0.058 s
14/11/15 23:46:41 INFO SparkContext: Job finished: first at
<console>:15, took 0.088417 s
res0: String = 0 0.64 0.64 0 0.32 0 0 0 0 0 0 0.64 0 0 0 0.32 0 1.29
1.93 0 0.96 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0.778 0 0 3.756 61 278 1


scala> nums.first()
[...]
14/11/15 23:46:42 INFO DAGScheduler: Stage 1 (first at <console>:17)
finished in 0.008 s
14/11/15 23:46:42 INFO TaskSchedulerImpl: Removed TaskSet 1.0, whose
tasks have all completed, from pool
14/11/15 23:46:42 INFO SparkContext: Job finished: first at
<console>:17, took 0.01287 s
res1: Array[Double] = Array(0.0, 0.64, 0.64, 0.0, 0.32, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.64, 0.0, 0.0, 0.0, 0.32, 0.0, 1.29, 1.93, 0.0,
0.96, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.778, 0.0, 0.0, 3.756, 61.0, 278.0, 1.0)
```

Operators in Spark are divided into transformations and actions. Transformations are evaluated lazily. Spark just creates the RDD's lineage graph when you call a transformation like map. No actual work is done until an action is invoked on the RDD. Creating the RDD and the map functions are transformations. The `.first()` function is an action that forces execution.

So when we created the inFile, it really didn't do anything except for creating a variable and set up the pointers. Only when we call an action like `.first()` does Spark evaluate the transformations. As a result, even if we point the inFile to a non-existent directory, Spark will take it. But when we call `inFile.first()`, it will throw the `Input path does not exist:` error.

Using the Spark shell to run logistic regression

When you run a command and do not specify a left-hand side of the assignment (that is leaving out the `val x` of `val x = y`), the Spark shell will assign a default name (that is, `res[number]` to the value. Now that you have the data in a more usable format, try to do something cool with it! Use Spark to run logistic regression over the dataset, as shown here:

```
scala> import breeze.linalg.{Vector, DenseVector}
import breeze.linalg.{Vector, DenseVector}


scala> case class DataPoint(x: Vector[Double], y: Double)
defined class DataPoint


scala>


scala> def parsePoint(x: Array[Double]): DataPoint = {
     |        DataPoint(new DenseVector(x.slice(0,x.size-2)) , x(x.size-
1))
     |        }
parsePoint: (x: Array[Double])DataPoint


scala> val points = nums.map(parsePoint(_))
points: org.apache.spark.rdd.RDD[DataPoint] = MappedRDD[3] at map at
<console>:21
```

```
scala> import java.util.Random
import java.util.Random

scala> val rand = new Random(42)
rand: java.util.Random = java.util.Random@24c55bf5

scala> points.first()
14/11/15 23:47:19 INFO SparkContext: Starting job: first at <console>:25
[..]
14/11/15 23:47:20 INFO SparkContext: Job finished: first at <console>:25,
took 0.188923 s
res2: DataPoint = DataPoint(DenseVector(0.0, 0.64, 0.64, 0.0, 0.32, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.64, 0.0, 0.0, 0.0, 0.32, 0.0, 1.29, 1.93, 0.0,
0.96, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.778, 0.0, 0.0, 3.756, 61.0),1.0)

scala> var w = DenseVector.fill(nums.first.size-2){rand.nextDouble}
14/11/15 23:47:36 INFO SparkContext: Starting job: first at <console>:20
[..]
14/11/15 23:47:36 INFO SparkContext: Job finished: first at <console>:20,
took 0.010883 s
w: breeze.linalg.DenseVector[Double] = DenseVector(0.7275636800328681,
0.6832234717598454, 0.30871945533265976, 0.27707849007413665,
0.6655489517945736, 0.9033722646721782, 0.36878291341130565,
0.2757480694417024, 0.46365357580915334, 0.7829017787900358,
0.9193277828687169, 0.43649097442328655, 0.7499061812554475,
0.38656687435934867, 0.17737847790937833, 0.5943499108896841,
0.20976756886633208, 0.825965871887821, 0.17221793768785243,
0.5874273817862956, 0.7512804067674601, 0.5710403484148672,
0.5800248845020607, 0.752509948590651, 0.03141823882658079,
0.35791991947712865, 0.8177969308356393, 0.41768754675291875,
0.9740356814958814, 0.7134062578232291, 0.48057451655643435,
0.2916564974118041, 0.9498601346594666, 0.8204918233863466,
0.636644547856282, 0.3691214939418974, 0.36025487536613...
scala> val iterations = 100
iterations: Int = 100

scala> import scala.math._
import scala.math._
```

```
scala> for (i <- 1 to iterations) {
     |         val gradient = points.map(p =>
     |           p.x * (1 / (1 + exp(-p.y*(w dot p.x))) - 1) * p.y
     |         ).reduce(_ + _)
     |         w -= gradient
     |       }
14/11/15 23:48:49 INFO SparkContext: Starting job: reduce at <console>:37
14/11/15 23:48:49 INFO DAGScheduler: Got job 4 (reduce at <console>:37)
with 2 output partitions (allowLocal=false)
[…]

14/11/15 23:48:53 INFO DAGScheduler: Stage 103 (reduce at <console>:37)
finished in 0.024 s
14/11/15 23:48:53 INFO SparkContext: Job finished: reduce at
<console>:37, took 0.027829 s


scala> w
res5: breeze.linalg.DenseVector[Double] = DenseVector(0.7336269947556883,
0.6895025214435749, 0.4721342862007282, 0.27723026762411473,
0.7829698104387295, 0.9109178772078957, 0.4421282714160576,
0.305394995185795, 0.4669066877779788, 0.8357335159675405,
0.9326548346504113, 0.5986886716855019, 0.7726151240395974,
0.3898162675706965, 0.18143939819778826, 0.8501243079114542,
0.28042415484918654, 0.867752122388921, 2.8395263204719647,
0.5976683218335691, 1.0764145195987342, 0.5718553843530828,
0.5876679823887092, 0.7609997638366504, 0.0793768969191899,
0.4177180953298126, 0.8177970052737001, 0.41885534550137715,
0.9741059468651804, 0.7137870996096644, 0.48057587402871155,
0.2916564975512847, 0.9533675296503782, 0.8204918691826701,
0.6367663765600675, 0.3833218016601887, 0.36677476558721556,...
scala>
```

If things went well, you were successful in using Spark to run logistic regression. That's awesome! We have just done a number of things; we defined a class and created an RDD and a function. As you can see, the Spark shell is quite powerful. Much of the power comes from it being based on the Scala REPL(the Scala interactive shell), and so it inherits all of the power of the Scala REPL. That being said, most of them time you will probably prefer to work with more traditional compiled code rather than in the REPL.

Interactively loading data from S3

Now try another exercise with the Spark shell. As part of Amazon's EMR Spark support, they have handily provided some sample data of Wikipedia traffic statistics in S3 in the format that Spark can use. To access the data, you first need to set your AWS access credentials as shell params. For instructions on signing up for EC2 and setting up the shell parameters, see *Running Spark on EC2* section in *Chapter 1, Installing Spark and Setting up your Cluster* (S3 access requires additional keys such as, `fs.s3n.awsAccessKeyId/awsSecretAccessKey` or using the `s3n://user:pw@` syntax). You can also set the shell parameters as `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`. We will leave the AWS configuration out of this discussion, but it needs to be completed. Once this is done, load the S3 data and take a look at the first line:

```
scala> val file = sc.textFile("s3n://bigdatademo/sample/wiki/")
14/11/16 00:02:43 INFO MemoryStore: ensureFreeSpace(34070) called with
curMem=512470, maxMem=278302556
14/11/16 00:02:43 INFO MemoryStore: Block broadcast_105 stored as values
in memory (estimated size 33.3 KB, free 264.9 MB)
file: org.apache.spark.rdd.RDD[String] = s3n://bigdatademo/sample/wiki/
MappedRDD[105] at textFile at <console>:17

scala> file.first()
14/11/16 00:02:58 INFO BlockManager: Removing broadcast 104
14/11/16 00:02:58 INFO BlockManager: Removing block broadcast_104
[..]
14/11/16 00:03:00 INFO SparkContext: Job finished: first at <console>:20,
took 0.442788 s
res6: String = aa.b Pecial:Listusers/sysop 1 4695

scala> file.take(1)
14/11/16 00:05:06 INFO SparkContext: Starting job: take at <console>:20
14/11/16 00:05:06 INFO DAGScheduler: Got job 105 (take at <console>:20)
with 1 output partitions (allowLocal=true)
14/11/16 00:05:06 INFO DAGScheduler: Final stage: Stage 105(take at
<console>:20)
[…]
14/11/16 00:05:07 INFO SparkContext: Job finished: take at <console>:20,
took 0.777104 s
res7: Array[String] = Array(aa.b Pecial:Listusers/sysop 1 4695)
```

You don't need to set your AWS credentials as shell params; the general form of the S3 path is `s3n://<AWS ACCESS ID>:<AWS SECRET>@bucket/path`.

It is important to take a look at the first line of the data; the reason for this is that Spark won't actually bother to load the data unless we force it to materialize something with it. It is useful to note that Amazon had provided a small sample dataset to get started with. The data is pulled from a much larger set available at `http://aws.amazon.com/datasets/4182`. This practice can be quite useful when developing in interactive mode as you want fast feedback of your jobs that are completing quickly. If your sample data was too big and your runs were taking too long, you could quickly slim down the RDD by using the `sample` functionality built into the Spark shell:

```
scala> val seed  = (100*math.random).toInt

seed: Int = 8

scala> val sample = file.sample(false,1/10.,seed)

res10: spark.RDD[String] = SampledRDD[4] at sample at <console>:17
```

If you wanted to rerun on the sampled data later, you could write it back to S3:

```
scala> sample.saveAsTextFile("s3n://mysparkbucket/test")

13/04/21 22:46:18 INFO spark.PairRDDFunctions: Saving as hadoop file
of type (NullWritable, Text)

....

13/04/21 22:47:46 INFO spark.SparkContext: Job finished:
saveAsTextFile at <console>:19, took 87.462236222 s
```

Now that you have the data loaded, find the most popular articles in a sample. First, parse the data by separating it into name and count. Then, reduce by the key summing the counts as there can be multiple entries with the same name. Finally, we swap the key/value so that when we sort by key, we get back the highest count item:

```
scala> val parsed = file.sample(false,1/10.,seed).map(x => x.split("
")).map(x => (x(1), x(2).toInt))

parsed: spark.RDD[(java.lang.String, Int)] = MappedRDD[5] at map at
<console>:16


scala> val reduced = parsed.reduceByKey(_+_)

13/04/21 23:21:49 WARN util.NativeCodeLoader: Unable to load native-
hadoop library for your platform... using builtin-java classes where
applicable

13/04/21 23:21:49 WARN snappy.LoadSnappy: Snappy native library not
loaded

13/04/21 23:21:50 INFO mapred.FileInputFormat: Total input paths to
process : 1
```

```
reduced: spark.RDD[(java.lang.String, Int)] = MapPartitionsRDD[8] at
reduceByKey at <console>:18


scala> val countThenTitle = reduced.map(x => (x._2, x._1))
countThenTitle: spark.RDD[(Int, java.lang.String)] = MappedRDD[9] at
map at <console>:20


scala> countThenTitle.sortByKey(false).take(10)
13/04/21 23:22:08 INFO spark.SparkContext: Starting job: take at
<console>:23
....
13/04/21 23:23:15 INFO spark.SparkContext: Job finished: take at
<console>:23, took 66.815676564 s
res1: Array[(Int, java.lang.String)] = Array((213652,Main_Page),
(14851,Special:Search), (9528,Special:Export/Can_You_Hear_Me),
(6454,Wikipedia:Hauptseite), (4189,Special:Watchlist),
(3520,%E7%89%B9%E5%88%A5:%E3%81%8A%E3%81%BE%E3%81%8B%E3%81%9B%E8%A1%A
8%E7%A4%BA), (2857,Special:AutoLogin), (2416,P%C3%A1gina_principal),
(1990,Survivor_(TV_series)), (1953,Asperger_syndrome))
```

Running Spark shell in Python

If you are more comfortable with Python than Scala, you can also work with Spark interactively in Python by running `[cmd]./pyspark[/cdm]`. Just to start working in the Python shell, let's perform the commands in quick start, as shown at `http://spark.apache.org/docs/1.1.0/quick-start.html`. This is just a simple exercise. We will see more of Python in *Chapter 9*, *Machine Learning Using Spark Mllib*:

```
$ bin/pyspark
[..]
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 1.1.1
      /_/

Using Python version 2.7.8 (default, Aug 21 2014 15:21:46)
SparkContext available as sc.
Let us read in a file
```

```
>>> textFile = sc.textFile("README.md")
14/11/16 00:12:11 INFO MemoryStore: ensureFreeSpace(34046) called with
curMem=0, maxMem=278302556
14/11/16 00:12:11 INFO MemoryStore: Block broadcast_0 stored as values in
memory (estimated size 33.2 KB, free 265.4 MB)
>>> textFile.count()
[..]
14/11/16 00:12:23 INFO DAGScheduler: Stage 0 (count at <stdin>:1)
finished in 0.733 s
14/11/16 00:12:23 INFO SparkContext: Job finished: count at <stdin>:1,
took 0.769692 s
141
>>> textFile.first()
14/11/16 00:12:35 INFO SparkContext: Starting job: runJob at PythonRDD.
scala:300
[..]
14/11/16 00:12:35 INFO SparkContext: Job finished: runJob at PythonRDD.
scala:300, took 0.029673 s
u'# Apache Spark'
>>> linesWithSpark = textFile.filter(lambda line: "Spark" in line)
>>> textFile.filter(lambda line: "Spark" in line).count()
14/11/16 00:13:15 INFO SparkContext: Starting job: count at <stdin>:1
[..]
14/11/16 00:13:15 INFO SparkContext: Job finished: count at <stdin>:1,
took 0.0379 s
21
>>>
```

As you can see, the Python operations are very similar to those in Scala.

Summary

In this chapter, you learned how to start the Spark shell and load our data, and you also did some simple machine learning. Now that you've seen how Spark's interactive console works, it's time to see how to build Spark jobs in a more traditional and persistent environment in the subsequent chapters.

3
Building and Running a Spark Application

Using Spark in an interactive mode with the Spark shell has limited permanence and does not work in Java. Building Spark jobs is a bit trickier than building a normal application as all dependencies have to be available on all the machines that are in your cluster. This chapter will cover the process of building a Java and Scala Spark job with Maven or **sbt** (**simple-build-tool**) and how to build Spark jobs with a non-Maven aware build system. A reference website to build Spark is `http://spark.apache.org/docs/latest/building-spark.html`.

Building your Spark project with sbt

The sbt is a popular build tool for Scala that supports building both Scala and Java codes. Building Spark projects with sbt is one of the easiest options. Spark release was originally built with sbt, but now they use Maven. However, the various members of the team actively use both sbt and Maven. The current normal method of building packages that use sbt is to use a shell script that bootstraps the specific version of sbt your project uses, thus making installation simpler.

If you are using a prebuilt Spark version, you will need to download and create the `sbt` directory.

As a first step, take a Spark job that already works and go through the process of creating a build file for it. In the Spark directory, start by copying the `GroupByTest` example into a new directory, as shown here:

```
mkdir -p example-scala-build/src/main/scala/spark/examples/

cp -af sbt example-scala-build//

cp examples/src/main/scala/org/apache/spark/examples/GroupByTest.scala
example-scala-build/src/main/scala/spark/examples/
```

As you are going to ship your JAR to the other machines, you will want to ensure all dependencies are included. You can either add a bunch of JARs or use a handy sbt plugin called `sbt-assembly` to group everything into a single JAR. If you don't have a bunch of transitive dependencies, you may decide that using the assembly extension isn't for your project. Instead of using `sbt-assembly`, you probably want to run `sbt/sbt assembly` in the Spark project and add the resulting JAR, `core/target/ spark-core_2.10-1.1.1.jar`, to your class path. The `sbt assembly` package is a great tool to avoid manual management of a large number of JARs. To add the assembly extension to your build, add the following code to `project/plugins.sbt`:

```
resolvers += Resolver.url("artifactory",
url("http://scalasbt.artifactoryonline.com/scalasbt/sbt-plugin-
releases"))(Resolver.ivyStylePatterns)

resolvers += "Typesafe Repository" at
"http://repo.typesafe.com/typesafe/releases/"

resolvers += "Spray Repository" at "http://repo.spray.cc/"
addSbtPlugin("com.eed3si9n" % "sbt-assembly" % "0.8.7")
```

For sbt 0.13.6+, add `sbt-assembly` as a dependency in `project/assembly.sbt`:

```
addSbtPlugin("com.eed3si9n" % "sbt-assembly" % "0.12.0")
```

Resolvers are used by sbt so that it can find out where a package is; you can think of this as similar to specifying an additional apt **Personal Package Archive** (**PPA**) source with the exception that it only applies to the one package you are trying to build. If you load up the resolver URLs in your browser, most of them have the directory listing turned on, and so you can see what packages are provided by the resolver. These resolvers point at web URLs, but there are also resolvers available for local paths that can be useful during development. The `addSbt` plugin directive is deceptively simple; it tells the user to include the `sbt-assembly` package from `com.eed3si9n` in Version 0.8.7 and implicitly adds the Scala Version and the sbt Version. Make sure to run the sbt reload clean update to install new plugins.

Here is the build file for one of the examples of `GroupByTest.scala` as if it was being built on its own; put the following code in `./build.sbt`:

```
//Next two lines only needed if you decide to use the assembly plugin
import AssemblyKeys._assemblySettings

scalaVersion := "2.10.4"

name := "groupbytest"

libraryDependencies ++= Seq(
   "org.spark-project" % "spark-core_2.10" % "1.1.0"
)
```

If the preceding code does not work, you can try:

```
libraryDependencies += "org.apache.spark" %% "spark-
core" % "1.1.0"
```

Otherwise, you can try this code snippet:

```
libraryDependencies += "org.apache.spark" %% "spark-
core" % "1.1.1".
```

```
resolvers ++= Seq(
   "JBoss Repository" at
"http://repository.jboss.org/nexus/content/repositories/releases/",
   "Spray Repository" at "http://repo.spray.cc/",
   "Cloudera Repository" at "https://repository.cloudera.com/
artifactory/cloudera-repos/",
  "Akka Repository" at "http://repo.akka.io/releases/",
   "Twitter4J Repository" at "http://twitter4j.org/maven2/"
)
// Only include if using assembly
mergeStrategy in assembly <<= (mergeStrategy in assembly) { (old)
=>

  {
    case PathList("javax", "servlet", xs @ _*) =>
    MergeStrategy.first
    case PathList("org", "apache", xs @ _*) => MergeStrategy.first
    case "about.html"  => MergeStrategy.rename
    case x => old(x)

  }

}
```

As you can see, the build file is similar to `plugin.sbt` in format. There are a few unique things about this build file that are worth mentioning. Just like with the plugin file, you need to add a number of resolvers here so that sbt can find all the dependencies. Note that we are including it as `"org.spark-project" % "spark-core_2.10.4" % "1.1.0"` rather than using `"org.spark-project" %% "spark-core" % "1.1.0"`. If possible, you should try to use the `%%` format, which automatically adds the Scala version. Another unique part of this build file is the use of `MergeStrategy`. As multiple dependencies can define the same files, when you merge everything into a single JAR you need to tell the plugin how to handle it. It is a fairly simple build file other than the merge strategy you need to manually specify the Scala version of Spark you are using.

Note: If you have a different JDK on the master than JRE on the workers, you may want to switch the target JDK by adding the following to your build file:

```
javacOptions ++= Seq("-target", "1.6")
```

Now that your build file is defined, build your `GroupByTest` Spark job using the following command:

```
sbt/sbt clean compile package
```

It will then produce `target/scala-2.10.4/groupbytest_2.10.4-0.1-SNAPSHOT.jar`.

Run `sbt/sbt assembly` in the Spark directory to make sure you have the Spark assembly available to your class paths. The example requires a pointer to the location where Spark is using `SPARK_HOME;` provide a pointer to the example of JAR with `SPARK_EXAMPLES_JAR` for Spark to distribute out. We also need to specify the class path that we built to Scala locally with `-cp`. So we can then run the following example:

```
SPARK_HOME="../"  SPARK_EXAMPLES_JAR="./target/scala-
2.10.4/groupbytest-assembly-0.1-SNAPSHOT.jar"  scala -cp
/users/sparkuser/spark-1.1.0/example-scala-build/target/scala-
2.10.4/groupbytest_2.10.4-0.1-SNAPSHOT.jar:/users/sparkuser/spark-
1.1.0/core/target/spark-core-assembly-1.1.0.jar
spark.examples.GroupByTest local[1]
```

If you have decided to build all of your dependencies into a single JAR with the assembly plugin, we need to call it using this command:

```
sbt/sbt assembly
```

This will produce an assembly snapshot at `target/scala-2.10.4/groupbytest-assembly-0.1-SNAPSHOT.jar`, which you can then run in a very similar manner, simply without the `spark-core-assembly`, as shown here:

```
SPARK_HOME="../" \ SPARK_EXAMPLES_JAR="./target/scala-
2.10.4/groupbytest-assembly-0.1-SNAPSHOT.jar" \
 scala -cp /users/sparkuser/spark-1.1.0/example-scala-
build/target/scala-2.10.4/groupbytest-assembly-0.1-SNAPSHOT.jar
spark.examples.GroupByTest local[1]
```

You may run into merge issues with sbt assembly if things have changed; a quick search of the Web will probably provide better current guidance than anything that could be written in future. So you need to keep in mind future merge problems. In general, `MergeStategy.first` should work.

Your success in the preceding section may have given you a false sense of security. As sbt will resolve from the local cache, deps that were brought in by another project could mean that the code builds on one machine and not others. Delete your local ivy cache and run sbt clean to make sure. If some files fail to download, try looking at Spark's list of resolvers and add any missing ones to your `build.sbt`.

Building your Spark job with Maven

Maven is an open source Apache project that builds Spark jobs in Java or Scala. As of Version 1.2.0, the building Spark site states that Maven is the official recommendation for packaging Spark and is the "build of reference" too. As with sbt, you can include the Spark dependency through Maven central simplifying our build process. Also similar to sbt is the ability of Spark and all of our dependencies to put everything in a single JAR using a plugin or build Spark as a monolithic JAR using `sbt/sbt assembly` for inclusion.

To illustrate the build process for Spark jobs with Maven, this section will use Java as an example as Maven is more commonly used to build Java tasks. As a first step, let's take a Spark job that already works and go through the process of creating a build file for it. We can start by copying the `GroupByTest` example into a new directory and generating the Maven template, as shown here:

```
mkdir example-java-build/; cd example-java-build
mvn archetype:generate \
   -DarchetypeGroupId=org.apache.maven.archetypes \
   -DgroupId=spark.examples \
  -DartifactId=JavaWordCount \
  -Dfilter=org.apache.maven.archetypes:maven-archetype-quickstart
cp ../examples/src/main/java/spark/examples/JavaWordCount.java
JavaWordCount/src/main/java/spark/examples/JavaWordCount.java
```

Next, update your Maven `example-java-build/JavaWordCount/pom.xml` to include information on the version of Spark we are using. Also, since the example file we are working with requires a JDK version greater than 1.5, we will need to update the Java version that Maven is configured to use; the current version is 1.3. In between the project tags, we will need to add the following code:

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.spark-project</groupId>
    <artifactId>spark-core_2.10.4</artifactId>
    <version>1.1.0</version>
  </dependency>
</dependencies>
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <configuration>
        <source>1.7</source>
        <target>1.7</target>
      </configuration>
    </plugin>
  </plugins>
</build>
```

We can now build our JAR with the `mvn` package that can be run with the following command:

```
SPARK_HOME="../"  SPARK_EXAMPLES_JAR="./target/JavaWordCount-1.0-
SNAPSHOT.jar"  java -cp ./target/JavaWordCount-1.0-
SNAPSHOT.jar:../../core/target/spark-core-assembly-1.1.0.jar
spark.examples.JavaWordCount local[1] ../../README
```

As with sbt, we can use a plugin to include all of the dependencies in our JAR file. Between the `<plugins>` tags, add the following code:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-shade-plugin</artifactId>
  <version>2.3</version>
  <configuration>
    <!-- This transform is used so that merging of akka configuration
files works -->
    <transformers>
      <transformer implementation="org.apache.maven.plugins.shade.
resource.ApacheLicenseResourceTransformer">
      </transformer>
      <transformer implementation="org.apache.maven.plugins.shade.
resource.AppendingTransformer">
        <resource>reference.conf</resource>
      </transformer>
    </transformers>
  </configuration>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <goal>shade</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Then run `mvn assembly` and the resulting JAR can be run as shown in the preceding section; but leave out the Spark assembly JAR from the class path.

As I was writing this chapter (November 16, 2014), an e-mail chain crossed the Spark user group discussing sbt versus Maven. The use of Maven is recommended unless one needs some special capability of sbt.

Building your Spark job with something else

If neither sbt nor Maven suits your needs, you may decide to use another build system. Thankfully, Spark supports building a fat JAR with all its dependencies, which makes it easy to include in the build system of your choice. Simply run `sbt/sbt assembly` in the Spark directory and copy the resulting assembly JAR at `core/target/spark-core-assembly-1.1.0.jar` to your build dependencies, and you are good to go. It is more common to use the `spark-assembly-1.2.0-hadoop2.6.0.jar` file. These files exist in `$SPARK_HOME$/lib` (if users use a prebuilt version) or in `$SPARK_HOME$/ assembly/target/scala-2.10/` (if users build the source code with Maven or sbt).

No matter what your build system is, you may find yourself wanting to use a patched version of the Spark libraries. In this case, you can deploy your Spark library locally. I recommend giving it a different version number to ensure that sbt/Maven picks up the modified version. You can change the version by editing `project/SparkBuild.scala` and changing the `version:=` part according to the version you have installed. If you are using sbt, you should run `sbt/sbt update` in the project that is importing the custom version. For other build systems, you just need to ensure that you use the new assembly JAR as part of your build.

Some references are as follows:

- `http://spark.apache.org/docs/latest/building-spark.html`
- `http://www.scala-sbt.org/`
- `https://github.com/sbt/sbt-assembly`
- `http://spark-project.org/docs/latest/scala-programming-guide.html`
- `http://maven.apache.org/guides/getting-started/`
- `http://maven.apache.org/plugins/maven-compiler-plugin/examples/set-compiler-source-and-target.html`
- `http://maven.apache.org/plugins/maven-dependency-plugin/`

Summary

So now you can build your Spark jobs with Maven, sbt, or the build system of your choice. It's time to jump in and start learning how to do more fun and exciting things such as learning how to create a Spark context in the subsequent chapter.

4
Creating a SparkContext

This chapter will cover how to create a `SparkContext` object in your cluster. A `SparkContext` object represents the connection to a Spark cluster and provides the entry point to interact with Spark. We need to create `SparkContext` so that we can interact with Spark and distribute our jobs. In *Chapter 2*, *Using the Spark Shell*, we interacted with Spark through the Spark shell that created a `SparkContext` object. Now you can create RDDs, broadcast variables and counters, and actually do fun things with your data. The Spark shell serves as an example of interacting with the Spark cluster through a `SparkContext` object in `./spark/repl/Main.scala`, as shown here:

```
def createSparkContext(): SparkContext = {
    val master = this.master match {
      case Some(m) => m
      case None => {
        val prop = System.getenv("MASTER")
        if (prop != null) prop else "local"
      }
    }
    sparkContext = new SparkContext(master, "Spark shell")
    sparkContext
  }
```

The preceding code snippet creates a `SparkContext` object using the provided `MASTER` environment variable (or local if none are set) called `Spark Shell` and doesn't specify any dependencies. This is because `Spark Shell` is built into Spark, and as such it doesn't have any JARs that needs to be distributed.

For a client to establish a connection to the Spark cluster, the `SparkContext` object needs some basic information, which is given here:

- Master URL: Can be `local[n]` for local mode or `Spark://[sparkip]` for Spark Server or `mesos://path` for a Mesos cluster
- `application name`: This is a human-readable application name
- `sparkHome`: This is the path to Spark on the master/workers
- `jars`: This is the path to the JARs required for your job

Scala

In a Scala program, you can create a `SparkContext` object with the following code:

```
val sparkContext = new SparkContext(master_path, "application
name", ["optional spark home path"],["optional list of jars"])
```

While you can hardcode all of these values, it's better to read them from the environment with reasonable defaults. This approach provides maximum flexibility to run the code in a changing environment without having to recompile. Using local as the default value for the master makes it easy to launch your application in a test environment locally. By carefully selecting the defaults, you can avoid having to over specify this. Here is an example of it:

```
import spark.sparkContext
import spark.sparkContext._
import scala.util.Properties

val master = Properties.envOrElse("MASTER","local")
val sparkHome = Properties.get("SPARK_HOME")
val myJars = Seq(System.get("JARS"))
val sparkContext = new SparkContext(master, "my app", sparkHome,
myJars)
```

Java

To create a `SparkContext` object in Java, try using the following code:

```
import spark.api.java.JavaSparkContext;
…
JavaSparkContext ctx = new JavaSparkContext("master_url",
"application name", ["path_to_spark_home", "path_to_jars"]);
```

While the preceding code snippet works (once you have replaced the parameters with the correct values for your setup), it requires a code change if you change any of the parameters. So instead of that, you can use reasonable defaults and allow them to be overridden in a similar way to the following example of the Scala code:

```
String master = System.getEnv("MASTER");
if (master == null) {
    master = "local";
}
String sparkHome = System.getEnv("SPARK_HOME");
if (sparkHome == null) {
    sparkHome = "./";
}
String jars = System.getEnv("JARS");
JavaSparkContext ctx = new
JavaSparkContext(System.getenv("MASTER"), "my Java app",
System.getenv("SPARK_HOME"), System.getenv("JARS"));
```

SparkContext – metadata

The `SparkContext` object has a set of metadata that I found useful. The version number, application name, and memory available are useful. At the start of a Spark program, I usually display/log the version number.

Value	**Use**
`appName`	This is the application name. If you have established a convention, this field can be useful at runtime.
`getConf`	It returns configuration information.
`getExecutorMemoryStatus`	This retrieves the memory details. It could be useful if you want to check memory details. As Spark is distributed, the values do not mean that you are out of memory.
`Master`	This is the name of the master.
`Version`	I found this very useful – especially while testing with different versions.

Execute the following command from shell:

```
bin/spark-shell
scala> sc.version
res1: String = 1.1.1
```

As you can see, I am running Version 1.1.1:

```
scala> sc.appName
res2: String = Spark shell

scala> sc.master
res3: String = local[*]

scala> sc.getExecutorMemoryStatus
res4: scala.collection.Map[String,(Long, Long)] = Map(10.0.1.3:56814
-> (278302556,278302556))
```

The `10.0.1.3` is the address of the machine. The first value is the maximum amount of memory allocated for the block manager (for buffering the intermediate data or caching RDDs), while the second value is the amount of remaining memory:

```
scala> sc.getConf
res5: org.apache.spark.SparkConf = org.apache.spark.SparkConf@7bc17541

scala> sc.getConf.toString()
res6: String = org.apache.spark.SparkConf@48acaa84

scala>
```

A more informative call of this is given here:

```
scala> sc.getConf.toDebugString
res1: String =
spark.app.id=local-1422768546091
spark.app.name=Spark shell
spark.driver.host=10.0.1.3
spark.driver.port=51904
spark.executor.id=driver
spark.fileserver.uri=http://10.0.1.3:51905
spark.home=/usr/local/spark
spark.jars=
spark.master=local[*]
spark.repl.class.uri=http://10.0.1.3:51902
spark.tachyonStore.folderName=spark-237294fa-1a29-4550-b033-
9a73a8222774
```

Shared Java and Scala APIs

Once you have a `SparkContext` object created, it will serve as your main entry point. In the next chapter, you will learn how to use our `SparkContext` object to load and save data. You can also use `SparkContext` to launch more Spark jobs and add or remove dependencies. Some of the nondata-driven methods you can use on the `SparkContext` object are shown here:

Method	Use
`addJar(path)`	This adds the JAR for all future jobs run through the `SparkContext` object.
`addFile(path)`	This downloads the file to all nodes on the cluster.
`stop()`	This shuts down `SparkContext`.
`clearFiles()`	This removes the files so that new nodes will not download them.
`clearJars()`	This removes the JARs from being required for future jobs.

Python

The Python `SparkContext` object is a bit different than the Scala and Java contexts as Python doesn't use JARs to distribute dependencies. As you are still likely to have dependencies, set `pyFiles` with a ZIP file containing all the dependent libraries as desired on `SparkContext` (or leave it empty if you don't have any files to distribute). Create a Python `SparkContext` object using this code:

```
from pyspark import SparkContext

sc = SparkContext("master","my python app", sparkHome="sparkhome",
pyFiles="placeholderdeps.zip")
```

The context metadata from Python is similar to that in Spark, as shown here:

```
bin/pyspark


Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 1.1.1
      /_/


Using Python version 2.7.8 (default, Aug 21 2014 15:21:46)
SparkContext available as sc.
```

```
>>> sc.version
u'1.1.1'
>>> sc.appName
u'PySparkShell'
>>> sc.master
u'local[*]'
>>> sc.getExecutorMemoryStatus
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'SparkContext' object has no attribute
'getExecutorMemoryStatus'
>>> from pyspark.conf import SparkConf
>>> conf = SparkConf()
>>> conf.toDebugString()
u'spark.app.name=pyspark-shell\nspark.master=local[*]\nspark.submit.
pyFiles='
>>>
```

PySpark does not have the `getExecutorMemoryStatus` call yet but we can get some information with the `.toDebugString` call.

Now that you are able to create a connection with your Spark cluster, it's time to start loading our data into Spark.

Some more information is as follows:

- `http://spark-project.org/docs/latest/quick-start.html`
- `http://www-stat.stanford.edu/~tibs/ElemStatLearn/data.html`
- `https://github.com/mesos/spark/blob/master/repl/src/main/scala/spark/repl/SparkILoop.scala`
- `http://spark.apache.org/docs/latest/api/python/pyspark.context.SparkContext-class.html`
- `http://www.scala-lang.org/api/current/index.html#scala.util.Properties$`
- `http://spark.apache.org/docs/latest/api/java/org/apache/spark/SparkContext.html`

Summary

In this chapter, we covered how to connect to our Spark cluster using a `SparkContext` object. By using this knowledge, we will look at the different data sources we can use to load data into Spark in the next chapter.

5 Loading and Saving Data in Spark

By this point in the book, you have already experimented with the Spark shell, figured out how to create a connection with the Spark cluster, and built jobs for deployment. Now to make those jobs useful, you will learn how to load and save data in Spark. Spark's primary unit for representation of data is an RDD, which allows for easy parallel operations on the data. Other forms of data, such as counters, have their own representation. Spark can load and save RDDs from a variety of sources.

RDDs

Spark RDDs can be created from any supported Hadoop source. Native collections in Scala, Java, and Python can also serve as the basis for an RDD. Creating RDDs from a native collection is especially useful for testing.

Before jumping into the details on the supported data sources/links, take some time to learn about what RDDs are and what they are not. It is crucial to understand that even though an RDD is defined, it does not actually contain data but just creates the pipeline for it. (As an RDD follows the principle of lazy evaluation, it evaluates an expression only when it is needed, that is, when an action is called for.) This means that when you go to access the data in an RDD, it could fail. The computation to create the data in an RDD is only done when the data is referenced by caching or writing out the RDD. This also means that you can chain a large number of operations and not have to worry about excessive blocking in a computational thread. It's important to note during application development that you can write code, compile it, and even run your job; unless you materialize the RDD, your code may not have even tried to load the original data.

Each time you materialize an RDD, it is recomputed; if we are going to be using something frequently, a performance improvement can be achieved by caching the RDD.

Loading data into an RDD

Now the chapter will examine the different sources you can use for your RDD. If you decide to run it through the examples in the Spark shell, you can call `.cache()` or `.first()` on the RDDs you generate to verify that it can be loaded. In *Chapter 2, Using the Spark Shell*, you learned how to load data text from a file and from S3. In this chapter, we will look at different formats of data (text file and CSV) and the different sources (filesystem, HDFS) supported.

One of the easiest ways of creating an RDD is taking an existing Scala collection and converting it into an RDD. The `SparkContext` object provides a function called `parallelize` that takes a Scala collection and turns it into an RDD over the same type as the input collection, as shown here:

- Scala:

  ```
  val dataRDD = sc.parallelize(List(1,2,4))
  dataRDD.take(3)
  ```

- Java:

  ```
  import java.util.Arrays;
  import org.apache.spark.SparkConf;
  import org.apache.spark.api.java.*;
  import org.apache.spark.api.java.function.Function;

  public class LDSV01 {

    public static void main(String[] args) {
      // TODO Auto-generated method stub
      SparkConf conf = new SparkConf().setAppName("Chapter
      05").setMaster("local");
      JavaSparkContext ctx = new JavaSparkContext(conf);
      JavaRDD<Integer> dataRDD = ctx.parallelize(Arrays.
  asList(1,2,4));
      System.out.println(dataRDD.count());
      System.out.println(dataRDD.take(3));
    }

  }
  [..]
  ```

```
14/11/22 13:37:46 INFO SparkContext: Job finished: count at
Test01.java:13, took 0.153231 s
3
[..]
14/11/22 13:37:46 INFO SparkContext: Job finished: take at
Test01.java:14, took 0.010193 s
[1, 2, 4]
```

The reason for a full program in Java is that you can use the Scala and Python shell, but for Java you need to compile and run the program. I use Eclipse and add the JAR file `/usr/local/spark-1.1.1/assembly/target/scala-2.10/spark-assembly-1.1.1-hadoop2.4.0.jar` in the Java build path.

- Python:

```
rdd = sc.parallelize([1,2,3])
rdd.take(3)
```

The simplest method for loading external data is loading text from a file. This has a requirement that the file should be available on all the nodes in the cluster, which isn't much of a problem for local mode. When you're in a distributed mode, you will want to use Spark's `addFile` functionality to copy the file to all of the machines in your cluster. Assuming your `SparkContext` object is called `sc`, we could load text data from a file (you need to create the file):

- Scala:

```
import org.apache.spark.SparkFiles;
...
sc.addFile("spam.data")
val inFile = sc.textFile(SparkFiles.get("spam.data"))
inFile.first()
```

- Java:

```
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.*;
import org.apache.spark.SparkFiles;;

public class LDSV02 {

  public static void main(String[] args) {
    SparkConf conf = new SparkConf().setAppName("Chapter 05").
setMaster("local");
    JavaSparkContext ctx = new JavaSparkContext(conf);
    System.out.println("Running Spark Version : " +ctx.version());
    ctx.addFile("/Users/ksankar/fpds-vii/data/spam.data");
    JavaRDD<String> lines = ctx.textFile(SparkFiles.get("spam.
data"));
    System.out.println(lines.first());
  }
}
```

The runtime messages are interesting:

```
Running Spark Version : 1.1.1

<It copied the file to a temporary directory in the
cluster. This would work in local mode as well as in a
spark cluster of many machines>

14/11/22 14:05:43 INFO Utils: Copying
/Users/ksankar/Tech/spark/book/spam.data to
/var/folders/gq/70vnnyfj6913b6lms_td7gb40000gn/T/spark-
f4c60229-8290-4db3-a39b-2941f63aabf8/spam.data

14/11/22 14:05:43 INFO SparkContext: Added file
/Users/ksankar/Tech/spark/book/spam.data at
http://10.0.1.3:52338/files/spam.data with timestamp
1416693943289

14/11/22 14:05:43 INFO MemoryStore: ensureFreeSpace(163705)
called with curMem=0, maxMem=2061647216

14/11/22 14:05:43 INFO MemoryStore: Block broadcast_0
stored as values in memory (estimated size 159.9 KB, free
1966.0 MB)

14/11/22 14:05:43 INFO FileInputFormat: Total input paths
to process : 1

[..]

14/11/22 14:05:43 INFO SparkContext: Job finished: first at
Test02.java:13, took 0.191388 s

0 0.64 0.64 0 0.32 0 0 0 0 0 0 0.64 0 0 0 0.32 0 1.29 1.93
0 0.96 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0.778 0 0 3.756 61 278 1
```

- Python:

```
from pyspark.files import SparkFiles
…
sc.addFile("spam.data")
in_file = sc.textFile(SparkFiles.get("spam.data"))
in_file.take(1)
```

The resulting RDD is of the string type, with each line being a unique element in the RDD. `take(1)` is an action that picks the first element from the RDD.

Frequently, your input files will be CSV or TSV files, which you will want to read and parse and then create RDDs for processing. The two ways of reading CSV files are either reading and parsing them using our own functions or using a CSV library like `opencsv`.

Let's first look at parsing using our own functions:

- Scala:

```
val inFile = sc.textFile("Line_of_numbers.csv")
val numbersRDD = inFile.map(line => line.split(','))
scala> numbersRDD.take(10)
[..]
14/11/22 12:13:11 INFO SparkContext: Job finished: take at
<console>:18, took 0.010062 s
res7: Array[Array[String]] = Array(Array(42, 42, 55, 61, 53, 49,
43, 47, 49, 60, 68, 54, 34, 35, 35, 39))
It is an array of String. We need float or double
val numbersRDD = inFile.map(line => line.split(',')).map(_.
toDouble)
scala> val numbersRDD = inFile.map(line => line.split(',')).map(_.
toDouble)
<console>:15: error: value toDouble is not a member of
Array[String]
       val numbersRDD = inFile.map(line => line.split(',')).map(_.
toDouble)
This will not work as we have an array of array of strings. This
is where flatMap comes handy!
scala> val numbersRDD = inFile.flatMap(line => line.split(',')).
map(_.toDouble)
numbersRDD: org.apache.spark.rdd.RDD[Double] = MappedRDD[10] at
map at <console>:15

scala> numbersRDD.collect()
 [..]
res10: Array[Double] = Array(42.0, 42.0, 55.0, 61.0, 53.0, 49.0,
43.0, 47.0, 49.0, 60.0, 68.0, 54.0, 34.0, 35.0, 35.0, 39.0)
scala> numbersRDD.sum()
[..]
14/11/22 12:19:15 INFO SparkContext: Job finished: sum at
<console>:18, took 0.013293 s
res9: Double = 766.0
scala>
```

- Python:

```
inp_file = sc.textFile("Line_of_numbers.csv")
numbers_rdd = inp_file.map(lambda line: line.split(','))
>>> numbers_rdd.take(10)
```

```
[..]
14/11/22 11:12:25 INFO SparkContext: Job finished: runJob at
PythonRDD.scala:300, took 0.023086 s
[[u'42', u'42', u'55', u'61', u'53', u'49', u'43', u'47', u'49',
u'60', u'68', u'54', u'34', u'35', u'35', u'39']]
>>>
But we want the values as integers or double
numbers_rdd = inp_file.flatMap(lambda line: line.split(',')).
map(lambda x:float(x))
>>> numbers_rdd.take(10)
14/11/22 11:52:39 INFO SparkContext: Job finished: runJob at
PythonRDD.scala:300, took 0.022838 s
[42.0, 42.0, 55.0, 61.0, 53.0, 49.0, 43.0, 47.0, 49.0, 60.0]
>>> numbers_sum = numbers_rdd.sum()
[..]
14/11/22 12:03:16 INFO SparkContext: Job finished: sum at
<stdin>:1, took 0.026984 s
>>> numbers_sum
766.0
>>>
```

- Java:

```
import java.util.Arrays;
import java.util.List;

import org.apache.spark.SparkConf;
import org.apache.spark.api.java.*;
import org.apache.spark.api.java.function.DoubleFunction;
import org.apache.spark.api.java.function.FlatMapFunction;
import org.apache.spark.api.java.function.Function;
import org.apache.spark.api.java.function.Function2;
import org.apache.spark.SparkFiles;;

public class LDSV03 {

  public static void main(String[] args) {
    SparkConf conf = new SparkConf().setAppName("Chapter 05").
setMaster("local");
    JavaSparkContext ctx = new JavaSparkContext(conf);
    System.out.println("Running Spark Version : " +ctx.version());
    ctx.addFile("/Users/ksankar/fdps-vii/data/Line_of_numbers.
csv");
    //
    JavaRDD<String> lines = ctx.textFile(SparkFiles.get("Line_of_
numbers.csv"));
```

```
    //
    JavaRDD<String[]> numbersStrRDD = lines.map(new
Function<String,String[]>() {
      public String[] call(String line) {return line.split(",");}
    });
    List<String[]> val = numbersStrRDD.take(1);
    for (String[] e : val) {
      for (String s : e) {
        System.out.print(s+" ");
      }
      System.out.println();
    }
    //
    JavaRDD<String> strFlatRDD = lines.flatMap(new FlatMapFunction
<String,String>() {
      public Iterable<String> call(String line) {return Arrays.
asList(line.split(","));}
    });
    List<String> val1 = strFlatRDD.collect();
    for (String s : val1) {
      System.out.print(s+" ");
      }
    System.out.println();
    //
    JavaRDD<Integer> numbersRDD = strFlatRDD.map(new
Function<String,Integer>() {
      public Integer call(String s) {return Integer.parseInt(s);}
    });
    List<Integer> val2 = numbersRDD.collect();
    for (Integer s : val2) {
      System.out.print(s+" ");
      }
    System.out.println();
    //
    Integer sum = numbersRDD.reduce(new Function2<Integer,Integer,
Integer>() {
      public Integer call(Integer a, Integer b) {return a+b;}
    });
    System.out.println("Sum = "+sum);
  }
}
```

The results are as expected:

```
[..]
14/11/22 16:02:18 INFO AkkaUtils: Connecting to HeartbeatReceiver:
akka.tcp://sparkDriver@10.0.1.3:56479/user/HeartbeatReceiver
```

```
Running Spark Version : 1.1.1
14/11/22 16:02:18 INFO Utils: Copying /Users/ksankar/Tech/spark/
book/Line_of_numbers.csv to /var/folders/gq/70vnnyfj6913b6lms_
td7gb40000gn/T/spark-9a4bed6d-adb5-4e08-b5c5-5e9089d6e54b/Line_of_
numbers.csv
14/11/22 16:02:18 INFO SparkContext: Added file /Users/ksankar/
fdps-vii/data/Line_of_numbers.csv at http://10.0.1.3:56484/files/
Line_of_numbers.csv with timestamp
1416700938334
14/11/22 16:02:18 INFO MemoryStore: ensureFreeSpace(163705)
called with curMem=0, maxMem=2061647216
14/11/22 16:02:18 INFO MemoryStore: Block broadcast_0 stored
as values in memory (estimated size 159.9 KB, free 1966.0 MB)
14/11/22 16:02:18 INFO FileInputFormat: Total input paths to
process : 1
14/11/22 16:02:18 INFO SparkContext: Starting job: take at
Test03.java:25
[..]
14/11/22 16:02:18 INFO SparkContext: Job finished: take at
Test03.java:25, took 0.155961 s
42 42 55 61 53 49 43 47 49 60 68 54 34 35 35 39
14/11/22 16:02:18 INFO BlockManager: Removing broadcast 1
[..]
14/11/22 16:02:18 INFO SparkContext: Job finished: collect at
Test03.java:36, took 0.016938 s
42 42 55 61 53 49 43 47 49 60 68 54 34 35 35 39
14/11/22 16:02:18 INFO SparkContext: Starting job: collect at
Test03.java:45
[..]
14/11/22 16:02:18 INFO SparkContext: Job finished: collect at
Test03.java:45, took 0.016657 s
42 42 55 61 53 49 43 47 49 60 68 54 34 35 35 39
14/11/22 16:02:18 INFO SparkContext: Starting job: reduce at
Test03.java:51
[..]
14/11/22 16:02:18 INFO SparkContext: Job finished: reduce at
Test03.java:51, took 0.019349 s
Sum = 766
```

This also illustrates one of the ways of getting data out of Spark; you can transform it to a standard Scala array using the `collect()` function. The `collect()` function is especially useful for testing, in much the same way that the `parallelize()` function is. The `collect()` function collects the job's execution results, while `parallelize()` partitions the input data and makes it an RDD. The `collect` function only works if your data fits in memory in a single host (where your code runs on), and even in that case, it adds to the bottleneck that everything has to come back to a single machine.

The `collect()` function brings all the data to the machine that runs the code. So beware of accidentally doing `collect()` on a large RDD!

The `split()` and `toDouble()` functions doesn't always work out so well for more complex CSV files. `opencsv` is a versatile library for Java and Scala. For Python the CSV library does the trick. Let's use the `opencsv` library to parse the CSV files in Scala.

- Scala:

```
import au.com.bytecode.opencsv.CSVReader
import java.io.StringReader

sc.addFile("Line_of_numbers.csv")
val inFile = sc.textFile("Line_of_numbers.csv")
val splitLines = inFile.map(line => {
  val reader = new CSVReader(new StringReader(line))
  reader.readNext()
})
val numericData = splitLines.map(line =>
line.map(_.toDouble))
val summedData = numericData.map(row => row.sum)
println(summedData.collect().mkString(","))
[..]
14/11/22 12:37:43 INFO TaskSchedulerImpl: Removed TaskSet
13.0, whose tasks have all completed, from pool
14/11/22 12:37:43 INFO SparkContext: Job finished: collect
at <console>:28, took 0.0234 s
766.0
```

While loading text files into Spark is certainly easy, text files on local disk are often not the most convenient format for storing large chunks of data. Spark supports loading from all of the different Hadoop formats (sequence files, regular text files, and so on) and from all of the support Hadoop storage sources (HDFS, S3, HBase, and so on). You can also load your CSV into HBase using some of their bulk loading tools (like import TSV) and get your CSV data.

Sequence files are binary flat files consisting of key value pairs; they are one of the common ways of storing data for use with Hadoop. Loading a sequence file into Spark is similar to loading a text file, but you also need to let it know about the types of the keys and values. The types must either be subclasses of Hadoop's `Writable` class or be implicitly convertible to such a type. For Scala users, some natives are convertible through implicits in `WritableConverter`. As of Version 1.1.0, the standard `WritableConverter` types are int, long, double, float, boolean, byte arrays, and string. Let's illustrate by looking at the process of loading a sequence file of String to Integer, as shown here:

- Scala:

```
val data = sc.sequenceFile[String, Int](inputFile)
```

- Java:

```
JavaPairRDD<Text, IntWritable> dataRDD = sc.sequenceFile(file,
Text.class, IntWritable.class);
JavaPairRDD<String, Integer> cleanData = dataRDD.map(new
PairFunction<Tuple2<Text, IntWritable>, String, Integer>() {
 @Override
public Tuple2<String, Integer> call(Tuple2<Text, IntWritable>
pair) {
return new Tuple2<String, Integer>(pair._1().toString(),
pair._2().get());
}
});
```

Note that in the preceding cases, like with the text input, the file need not be a traditional file; it can reside on S3, HDFS, and so on. Also note that for Java, you can't rely on implicit conversions between types.

HBase is a Hadoop-based database designed to support random read/write access to entries. Loading data from HBase is a bit different from text files and sequence in files with respect to how we tell Spark what types to use for the data.

- Scala:

```
import spark._
import org.apache.hadoop.hbase.{HBaseConfiguration,
HTableDescriptor}
import org.apache.hadoop.hbase.client.HBaseAdmin
import org.apache.hadoop.hbase.mapreduce.TableInputFormat
….
val conf = HBaseConfiguration.create()
```

```
conf.set(TableInputFormat.INPUT_TABLE, input_table)
 // Initialize hBase table if necessary
val admin = new HBaseAdmin(conf)
if(!admin.isTableAvailable(input_table)) {
  val tableDesc = new HTableDescriptor(input_table)
  admin.createTable(tableDesc)
}
val hBaseRDD =  sc.newAPIHadoopRDD(conf,
      classOf[TableInputFormat],
      classOf[org.apache.hadoop.hbase.io.ImmutableBytesWritable],
      classOf[org.apache.hadoop.hbase.client.Result])
```

- Java:

```
import spark.api.java.JavaPairRDD;
import spark.api.java.JavaSparkContext;
import spark.api.java.function.FlatMapFunction;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.HTableDescriptor;
import org.apache.hadoop.hbase.client.HBaseAdmin;
import org.apache.hadoop.hbase.mapreduce.TableInputFormat;
import org.apache.hadoop.hbase.io.ImmutableBytesWritable;
import org.apache.hadoop.hbase.client.Result;
...
JavaSparkContext sc = new JavaSparkContext(args[0],
"sequence load", System.getenv("SPARK_HOME"),
System.getenv("JARS"));
Configuration conf = HBaseConfiguration.create();
conf.set(TableInputFormat.INPUT_TABLE, args[1]);
// Initialize hBase table if necessary
HBaseAdmin admin = new HBaseAdmin(conf);
if(!admin.isTableAvailable(args[1])) {
    HTableDescriptor tableDesc = new
HTableDescriptor(args[1]);
    admin.createTable(tableDesc);
}
JavaPairRDD<ImmutableBytesWritable, Result> hBaseRDD =
sc.newAPIHadoopRDD( conf, TableInputFormat.class,
ImmutableBytesWritable.class, Result.class);
```

The method that you used to load the HBase data can be generalized for loading all other sorts of Hadoop data. If a helper method in `SparkContext` does not already exist for loading the data, simply create a configuration specifying how to load the data and pass it into a new `APIHadoopRDD` function. Helper methods exist for plain text files and sequence files. A helper method also exists for Hadoop files similar to the Sequence file API.

Saving your data

While distributed computational jobs are a lot of fun, they are much more useful when the results are stored in a useful place. While the methods for loading an RDD are largely found in the `SparkContext` class, the methods for saving an RDD are defined on the RDD classes. In Scala, implicit conversions exist so that an RDD, that can be saved as a sequence file, is converted to the appropriate type, and in Java explicit conversion must be used.

Here are the different ways to save an RDD:

- For Scala:

  ```
  rddOfStrings.saveAsTextFile("out.txt")
  keyValueRdd.saveAsObjectFile("sequenceOut")
  ```

- For Java:

  ```
  rddOfStrings.saveAsTextFile("out.txt")
  keyValueRdd.saveAsObjectFile("sequenceOut")
  ```

- For Python:

  ```
  rddOfStrings.saveAsTextFile("out.txt")
  ```

> In addition, users can save the RDD as a compressed text file by using the following function:
>
> ```
> saveAsTextFile(path: String, codec: Class[_
> <: CompressionCodec])
> ```

Some references are as follows:

- `http://spark-project.org/docs/latest/scala-programming-guide.html#hadoop-datasets`
- `http://opencsv.sourceforge.net/`
- `http://commons.apache.org/proper/commons-csv/`
- `http://hadoop.apache.org/docs/current/api/org/apache/hadoop/mapred/SequenceFileInputFormat.html`
- `http://hadoop.apache.org/docs/current/api/org/apache/hadoop/mapred/InputFormat.html`
- `http://www.michael-noll.com/tutorials/running-hadoop-on-ubuntu-linux-single-node-cluster/`
- `http://spark.apache.org/docs/latest/api/python/`
- `http://wiki.apache.org/hadoop/SequenceFile`

- `http://hbase.apache.org/book/quickstart.html`
- `http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/mapreduce/TableInputFormat.html`
- `https://spark.apache.org/docs/latest/api/java/org/apache/spark/api/java/JavaPairRDD.html`
- `https://bzhangusc.wordpress.com/2014/06/18/csv-parser/`

Summary

In this chapter, you saw how to load data from a variety of different sources. We also looked at basic parsing of the data from text input files. Now that we can get our data loaded into a Spark RDD, it is time to explore the different operations we can perform on our data in the next chapter.

6
Manipulating your RDD

The last few chapters have been the necessary groundwork to get Spark working. Now that you know how to load and save your data in different ways, it's time for the big payoff, that is, manipulating data. The API to manipulate your RDD is similar among the languages but not identical. Unlike the previous chapters, each language is covered in its own section; you likely only need to read the one pertaining to the language you are interested in using. Particularly, the Python implementation is currently not fully at feature parity with the Scala/Java API, but it supports most of the basic functionality as of version 1.1.0 and has plans to improve feature parity in the future versions.

Manipulating your RDD in Scala and Java

RDDs are the primary abstraction in Spark. From a structural view, they are just a bunch of elements—but elements that can be operated in parallel!

Manipulating your RDD in Scala is quite simple, especially if you are familiar with Scala's collection library. Many of the standard functions are available directly on Spark's RDDs with the primary catch being that they are immutable. This makes porting existing Scala code to be distributed much simpler than porting Java or Python code. At least in theory, this is true. While Scala encourages functional programming, one can always use Scala in a non-functional way. Vice versa, while using Python, one can, to a large extent, apply a functional approach to programming. In other words, the difference lies in whether it is the functional/immutable style of programming or not, and the programs written in a functional way can be ported to Spark easily.

Manipulating your RDD in Java is fairly simple but a little more awkward at times than it is in Scala. There are a couple of reasons for this. The main reason has to do with **type inference** and also with the fact that Java doesn't have anonymous functions. In the following code snippets, sometimes the Java code is more unwieldy because Java lacks type inference and anonymous functions. Java 8 has **lambda**, which would make Java a lot more elegant with Spark. Secondly, as Java doesn't have implicit conversions, we have to be more explicit with our types. While the return types are Java friendly, Spark requires the use of Scala's `Tuple2` class for key-value pairs.

The hallmark of a MapReduce system are the two primitives: **map** and **reduce**. We've seen the map function used in the past chapters. Map works by taking in a function, which acts on each individual element in the input RDD and produces a new output element. For example, to produce a new RDD where you add one to every number, use `rdd.map(x => x+1).`

Alternatively, in Java, you can use this:

```
rdd.map(new Function<Integer, Integer>() { public Integer
  call(Integer x) { return x+1;} });
```

There are actually two types of map function—`map` and `flatMap`. It is easy to get confused between them. The `map` function takes an element and returns an element. The element could be a single entity, a tuple, or a list; nevertheless, there is a one-to-one correspondence with the `map` function. The `flatMap` function, on the other hand, takes one element and will return one or more elements. Actually, the `map` in Hadoop MapReduce is `flatMap`. In fact, the Spark word count example is implemented using the `flatMap()`, `map()`, and `reduceByKey()` functions.

It is important to understand that the `map` function and the other Spark functions do not modify/update the existing elements; rather, they return a new RDD with new elements—the RDDs are immutable. The `reduce` function takes a function that operates on pairs to combine all the data. The `reduce` function you provide needs to be commutative and associative (that is, *f(a,b) == f(b,a)*, and *f(a,f(b,c)) == f(f(a,b),c)*). For example, to sum all of the elements, you need to use `rdd.reduce(x,y => x+y)` or `rdd.reduce(new Function2<Integer, Integer>(){ public Integer call(Integer x, Integer y) { return x+y;} }`.

All functions are not commutative; for example, while multiplication is commutative 2*3 = 3*2, subtraction is not, that is, 3-2 is not the same as 2-3, and division is not, that is, 4/2 is not the same as 2/4. The same applies for associativity; sum is associative, that is, 2+3+4 = (2+3)+4 or 2+(3+4), but average is not, that is, average (2,3,4,5,6) is not equal to average (2,3) + average (4,5,6).

The `flatMap` function is a useful utility function that lets you write a function that returns an iterable of the type you want and then flattens the results. A simple example of this is a case where you want to parse all of the data, but some of it might fail to be parsed. The `flatMap` function can be used to output an empty list if it has failed or a list with success if it has worked. Another example when the output collection has a different size than the input collection is while parsing a document and splitting in words; here every line may contain one or more words.

In addition to the `reduce` function, there is a corresponding `reduceByKey` function that works on RDDs, which are key-value pairs to produce another RDD. Unlike when you're using map on a list in Scala, your function will run on a number of different machines, and so you can't depend on the shared state with this.

Before continuing into other wonderful functions for manipulating your RDD, you need to read a bit about shared states. In the example given earlier where we added one to every integer, we didn't really share states. However, for even simple tasks such as distributed parsing of data that we did when loading the CSV file, it can be quite handy to have shared counters for things such as keeping track of the number of rejected records. Spark supports both shared immutable data, which it calls **broadcast** and **accumulate** (via accumulators):

- You can create a new broadcast by calling `sc.broadcast(value)`. While you don't have to explicitly broadcast values as Spark does its magic in the background, broadcasting ensures that the value is sent to each node only once. Broadcasts are often used for things such as side inputs (for example, a hashmap that you need to look up as part of the `map` function). This returns an object that can be used to reference the broadcast value.
- Another method for sharding states is using an accumulator. To create an accumulator, use `sc.accumulator(initialvalue)`. This returns an object you can add to in a distributed context and then get back the value by calling `.value()`. The `accumulableCollection` can be used to create a collection that is appended in a distributed fashion; however, if you find yourself using this, ask yourself whether you could use the results of a map output instead. If the predefined accumulators don't work for your use case, you can use `accumulable` to define your own accumulation type. A broadcast value can be read by all of the workers and an accumulator can be written by all of the workers but read by only the driver.

If you are writing Scala code that interacts with a Java Spark process (say for testing), you may find it useful to use the `int` accumulator and similar others on the Java Spark context; otherwise, your accumulator types might not quite match up.

If you find that your accumulator isn't increasing in value like you expect, remember that Spark follows the principle of lazy evaluation. This means that Spark won't actually perform the maps, reductions, or other computation on RDDs until the data has to be output.

Look at the previous example, which parsed CSV files, and make it a bit more robust. In your previous work, you had assumed that the input was well formatted and if any errors occur, our entire pipeline would fail. While this can be the correct behavior for some kind of work, we may want to accept some number of malformed records while dealing with data from third parties. On the other hand, we don't want to just throw out all of the records and declare it a success; we might miss an important format change and produce meaningless results. Consider the following code:

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkFiles;

import au.com.bytecode.opencsv.CSVReader

import java.io.StringReader

object LoadCsvWithCountersExample {
  def main(args: Array[String]) {
    if (args.length != 2) {
      System.err.println("Usage: LoadCsvExample <master>
        <inputfile>")
      System.exit(1)
    }
    val master = args(0)
    val inputFile = args(1)
    val sc = new SparkContext(master, "Load CSV With Counters
Example",
                 System.getenv("SPARK_HOME"),
                 Seq(System.getenv("JARS")))
    val invalidLineCounter = sc.accumulator(0)
    val invalidNumericLineCounter = sc.accumulator(0)
    sc.addFile(inputFile)
    val inFile = sc.textFile(inputFile)
    val splitLines = inFile.flatMap(line => {
      try {
    val reader = new CSVReader(new StringReader(line))
      Some(reader.readNext())
      } catch {
    case _ => {
      invalidLineCounter += 1
      None
    }
      }
    }
            )
    val numericData = splitLines.flatMap(line => {
      try {
    Some(line.map(_.toDouble))
      } catch {
    case _ => {
      invalidNumericLineCounter += 1
      None
    }
      }
    }
    )
    val summedData = numericData.map(row => row.sum)
    println(summedData.collect().mkString(","))
    println("Errors: "+invalidLineCounter+","
      +invalidNumericLineCounter)
  }
}
```

You can run the code with parameters `local/path/Line_of_numbers.csv` and the code will run with the following result:

```
[..]
2014-11-22 18:15:48,399 INFO  [main] spark.SparkContext (Logging.
scala:logInfo(59)) - Job finished: collect at LoadCsvWithCountersExample.
scala:47, took 0.256383 s
766.0
Errors: 0,0
```

Alternatively, in Java you can do the following:

```
    import org.apache.spark.Accumulator;
    import org.apache.spark.api.java.JavaRDD;
    import org.apache.spark.api.java.JavaPairRDD;
    import org.apache.spark.api.java.JavaSparkContext;
    import org.apache.spark.api.java.function.FlatMapFunction;
```

```
import au.com.bytecode.opencsv.CSVReader;

import java.io.StringReader;
import java.util.Arrays;
import java.util.List;
import java.util.ArrayList;

public class JavaLoadCsvCounters {
  public static void main(String[] args) throws Exception {
    if (args.length != 2) {
    System.err.println("Usage: JavaLoadCsvCounters <master>
      <inputfile>");
    System.exit(1);
    }
    String master = args[0];
    String inputFile = args[1];
    JavaSparkContext sc = new JavaSparkContext(master, "java load
      csv with counters",
        System.getenv("SPARK_HOME"), System.getenv("JARS"));
    final Accumulator<Integer> errors = sc.accumulator(0);
    JavaRDD<String> inFile = sc.textFile(inputFile);
    JavaRDD<Integer[] > splitLines = inFile.flatMap(new
      FlatMapFunction<String, Integer[]> (){
        public Iterable<Integer[]> call(String line) {
        ArrayList<Integer[]> result = new ArrayList<Integer[]>();
        try {
            CSVReader reader = new CSVReader(new StringReader
              (line));
              String[] parsedLine = reader.readNext();
              Integer[] intLine = new Integer[parsedLine.length];
              for (int i = 0; i < parsedLine.length; i++) {
                intLine[i] = Integer.parseInt(parsedLine[i]);
              }
              result.add(intLine);
         } catch (Exception e) {
              errors.add(1);
         }
           return result;
         }
    }
    );
    List <Integer[]> res = splitLines.collect();
    System.out.print("Loaded data ");
    for (Integer[] e : res) {
      for (Integer val:e) {
        System.out.print(val+" ");
      }
      System.out.println();
    }
    System.out.println("Error count "+errors.value());
  }
}
```

You can run the code with parameters `local/path/Line_of_numbers.csv` and the code will run with the following result:

```
[..]
14/11/22 19:33:05 INFO SparkContext: Job finished: collect at
JavaLoadCsvCounters.java:44, took 0.106908 s
Loaded data 42 42 55 61 53 49 43 47 49 60 68 54 34 35 35 39
Error count 0
```

The preceding code example illustrates the usefulness of `flatMap`. In general, `flatMap` can be used when the required output collection is of a different size than that of the input collection. You can do this because in general there are nested collections or types involved, which need to be flattened. As the options in Scala can be used as sequences through an implicit conversion, you can avoid having to explicitly filter out the `None` result and just use `flatMap`.

Summary statistics can be quite useful when examining large datasets. In the preceding example, you loaded the data as `Doubles` to use Spark's provided summary statistics capabilities on the RDD. In Java, this requires explicitly using the `JavaDoubleRDD` type. For Java, it is important to use `DoubleFunction<Integer[]>` rather than `Function<Integer[], Double>` in the example as the second option won't result in the `JavaDoubleRDD` type. No such consideration is required for Scala as implicit conversions deal with the details. Compute the mean and the variance or compute them together with stats. You can extend this by adding it at the end of the preceding function to print out the summary statistics as `println(summedData.stats())`.

To do this with Java, we would do it as follows:

```
    JavaDoubleRDD summedData = splitLines.map(new
      DoubleFunction<Integer[]>() {
            public Double call(Integer[] in) {
              Double ret = 0.;
              for (int i = 0; i < in.length; i++) {
                ret += in[i];
              }
              return ret;
            }
    }
    );
    System.out.println(summedData.stats());
```

While working with key-value pair data, it can be quite useful to group data with the same key together (for example, if the key represents a user or a sample). The `groupByKey` function provides an easy way to group data together by key. The `groupByKey` function is a special case of `combineByKey`. There are several functions in the `PairRDD` class that are all implemented very closely on top of `combineByKey`. If you find yourself using `groupByKey` or one of the other functions derived from `combineByKey` and immediately transforming the result, you should check to see whether there is a function better suited to the task. A common thing to do while starting out is to perform `groupByKey` and then sum the results with `groupByKey().map({case (x,y) => (x,y.sum)})`. Alternatively, in Java you can do the following:

```
pairData.groupByKey().mapValues(new Function<List<Integer>,
  Integer >(){
        public Integer call(List<Integer> x) {
          Integer sum = 0;
          for (Integer i : x) {
            sum += i;
          }
          return sum;
        }
}
); or in python .map(lambda (x,y): (x,sum(y))).collect()
```

By using `reduceByKey`, it could be simplified to `reduceByKey((x,y) => x+y)` or in Java, as follows:

```
pairData.groupByKey().mapValues(
  new Function<Iterable<Integer>, Integer >(){
    public Integer call(Iterable<Integer> x) {
      Integer sum = 0; for (Integer i : x) {
        sum += i;
      }
    return sum;
    }
  }
);
```

In fact, this may be much more efficient. No big shuffle is needed, as is the case for the `groupBy`. The only thing required is an aggregation of the values, which is important.

The `foldByKey(zeroValue)(function)` function is similar to a traditional fold operation, which works per key. In a traditional fold, a list that is provided would be called with the initial value and the first element of the list, and then the resulting value and the next element of the list would be the input to the next call of fold. Doing this requires sequentially processing the entire list, and so `foldByKey` behaves slightly differently. There is a handy table of functions of PairRDDs at the end of this section.

Sometimes, you will only want to update the values of a key-value pair data structure such as a PairRDD. You've learned about `foldByKey` and how it doesn't quite work as a traditional fold. If you're a Scala developer and you require the "traditional" fold behavior, you can perform the `groupByKey` function and then map a fold by value over the resulting RDD. This is an example of a case where you only want to change the value and we don't care about the key of the RDD; so examine the following code:

```
rdd.groupByKey().mapValues(x => {x.fold(0)((a,b) => a+b)})
```

The preceding code is interesting as it combines the Spark function `groupByKey` with a Scala function `fold()`.The `groupBy()` function shuffles the data so that the values are "together". The fold mentioned is a "local" Scala fold on the nodes in parallel.

Often your data won't come in cleanly from a single source and you will want to join the data together for processing, which can be done with `coGroup`. This can be done when you are joining web access logs with transaction data or just joining two different computations on the same data. Provided that the RDDs have the same key, we can join two RDDs together with `rdd.coGroup(otherRdd)`. There are a number of different join functions for different purposes illustrated in the table at the end of this section.

The next task you will learn is distributing files among the cluster. We illustrate this by adding GeoIP support and mixing it together with the gradient descent example from the earlier chapter. Sometimes, the libraries you will use need files distributed along with them. While it is possible to add them to the JAR and access them as class objects, Spark provides a simple way to distribute the required files by calling `addFile()`, as shown here:

```
import scala.math

import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkFiles;
import org.apache.spark.util.Vector

import au.com.bytecode.opencsv.CSVReader
import java.util.Random
import java.io.StringReader
import java.io.File

import com.snowplowanalytics.maxmind.geoip.IpGeo

case class DataPoint(x: Vector, y: Double)

object GeoIpExample {

  def main(args: Array[String]) {
    if (args.length != 2) {
      System.err.println("Usage: GeoIpExample <master>
        <inputfile>")
      System.exit(1)
    }
    val master = args(0)
    val inputFile = args(1)
    val iterations = 100
    val maxMindPath = "GeoLiteCity.dat"
    val sc = new SparkContext(master, "GeoIpExample",
                 System.getenv("SPARK_HOME"),
                 Seq(System.getenv("JARS")))
    val invalidLineCounter = sc.accumulator(0)
    val inFile = sc.textFile(inputFile)
    val parsedInput = inFile.flatMap(line => {
      try {
        val row = (new CSVReader(new StringReader
          (line))).readNext()
          Some((row(0),row.drop(1).map(_.toDouble)))
      } catch {
        case _ => {
        invalidLineCounter += 1
        None
      }
      }
    })
    val geoFile = sc.addFile(maxMindPath)
    // getLocation gives back an option so we use flatMap to only
       output if its a some type
    val ipCountries = parsedInput.flatMapWith(_ => IpGeo(dbFile =
      SparkFiles.get(maxMindPath) ))((pair, ipGeo) => {
     ipGeo.getLocation(pair._1).map(c => (pair._1,
       c.countryCode)).toSeq
     })
```

```
      ipCountries.cache()
      val countries = ipCountries.values.distinct().collect()
      val countriesBc = sc.broadcast(countries)
      val countriesSignal = ipCountries.mapValues(country =>
        countriesBc.value.map(s => if (country == s) 1. else 0.))
      val dataPoints = parsedInput.join(countriesSignal).map(input
        => {
        input._2 match {
      case (countryData, originalData) => DataPoint(new
        Vector(countryData++originalData.slice(1,originalData.size-
        2)) , originalData(originalData.size-1))
        }
      })
      countriesSignal.cache()
      dataPoints.cache()
      val rand = new Random(53)
      var w = Vector(dataPoints.first.x.length, _ => rand.
        nextDouble)
      for (i <- 1 to iterations) {
        val gradient = dataPoints.map(p =>
      (1 / (1 + math.exp(-p.y*(w dot p.x))) - 1) * p.y * p.x).
        reduce(_ + _)
        w -= gradient
      }
      println("Final w: "+w)
    }
  }
```

In this example, you see multiple Spark computations. The first is to determine all of the countries where our data is; so we can map the country to a binary feature. The code then uses a public list of proxies and the reported latency to try and estimate the latency I measured. This also illustrates the use of `mapWith`. If you have a mapping job that needs to create a per partition resource, `mapWith` can be used to do this. This can be useful for connections to backends or the creation of something like a PRNG. Some elements also can't be serialized over the wire (such as the `IpCountry` in the example), and so you have to create them per shard. You can also see that we cache a number of our RDDs to keep them from having to be recomputed.

There are several options when working with multiple RDDs.

Scala RDD functions

These are PairRDD functions based on `combineByKey`. All operate on RDDs of type `[K,V]`:

Function	Param options	Explanation	Return type
`foldByKey`	`(zeroValue)(func(V,V)=>V)` `(zeroValue, partitioner)(func(V,V=>V)` `(zeroValue, partitions)(func(V,V=>V)`	`foldByKey` merges the values using the provided function. Unlike a traditional fold over a list, the `zeroValue` can be added an arbitrary number of times.	RDD[K,V]
`reduceByKey`	`(func(V,V)=>V)` `(func(V,V)=>V, numTasks)`	`reduceByKey` is the parallel version of reduce that merges the values for each key using the provided function and returns an RDD.	RDD[K,V]
`groupByKey`	`()` `(numPartitions)`	This groups elements together by key.	RDD[K,Seq[V]]

Functions for joining PairRDDs

Often while working with two or more key-value RDDs, it is useful to join them together. There are a few different methods to do this depending on what your desired behavior is:

Function	Param options	Explanation	Return type
`coGroup`	`(otherRdd[K,W]...)`	Join two (or more) RDDs by the shared key. Note if an element is missing in one RDD but present in the other one, the `Seq` value will simply be empty.	RDD[(K,(Seq[V], Seq[W]...))]
`join`	`(otherRdd[K,W])` `(otherRdd[K,W], partitioner)` `(otherRdd[K,W], numPartitions)`	Join an RDD with another RDD. The result is only present for elements where the key is present in both RDDs.	RDD[(K,(V,W))]
`subtractKey`	`(otherRdd[K,W])` `(otherRdd[K,W], partitioner)` `(otherRdd[K,W], numPartitions)`	This returns an RDD with only keys not present in the other RDD.	RDD[(K,V)]

Other PairRDD functions

Some functions only make sense when working on key-value pairs, as follows:

Function	Param options	Explanation	Return type
`lookup`	`(key: K)`	This looks up a specific element in the RDD. It uses the RDD's partitioner to figure out which shard(s) to look at.	Seq[V]
`mapValues`	`(f: V => U)`	This is a specialized version of map for PairRDDs when you only want to change the value of the key-value pair. It takes the provided `map` function and applies it to the value. If you need to make your change based on both key and value, you must use one of the normal RDD map functions.	RDD[(K,U)]
`collectAsMap`	`()`	This takes an RDD and returns a concrete map. Your RDD must be able to fit in memory.	Map[K, V]
`countByKey`	`()`	This counts the number of elements for each key.	Map[K, Long]
`partitionBy`	`(partitioner: Partitioner, mapSideCombine: Boolean)`	This returns a new RDD with the same data but partitioned by the new partitioner. The Boolean flag `mapSideCombine` controls whether Spark should group values with the same key together before repartitioning. It defaults to `false` and sets to `true` if you have a large percentage of duplicate keys.	RDD[(K,V)]

Function	**Param options**	**Explanation**	**Return type**
`flatMapValues`	`(f: V => TraversableOnce [U])`	This is similar to `MapValues`. It's a specialized version of `flatMap` for PairRDDs when you only want to change the value of the key-value pair. It takes the provided `map` function and applies it to the value. The resulting sequence is then "flattened", that is, instead of getting `Seq[Seq[V]]`, you get `Seq[V]`. If you need to make your change based on both key and value, you must use one of the normal RDD map functions.	RDD[(K,U)]

For information on saving PairRDDs, refer to the previous chapter.

Double RDD functions

Spark defines a number of convenience functions that work when your RDD is comprised of doubles, as follows:

Function	**Arguments**	**Return value**
`Mean`	`()`	Average
`sampleStdev`	`()`	Standard deviation for a sample rather than a population (as it divides by *N-1* rather than *N*).
`Stats`	`()`	Mean, variance, and count as a `StatCounter`.
`Stdev`	`()`	Standard deviation (for population).
`Sum`	`()`	Sum of the elements.
`variance`	`()`	Variance

General RDD functions

The remaining RDD functions are defined on all RDDs:

Function	Arguments	Returns
`aggregate`	`(zero: U)(seqOp: (U,T) => T, combOp (U, U) => U)`	It aggregates all of the elements of each partition of an RDD and then combines them using `combOp`. The zero value should be neutral (that is 0 for + and 1 for *).
`cache`	`()`	It caches an RDD reused without re-computing. It's the same as `persist(StorageLevel.MEMORY_ONLY)`.
`collect`	`()`	It returns an array of all of the elements in the RDD.
`count`	`()`	It returns the number of elements in an RDD.
`countByValue`	`()`	It returns a map of value to the number of times that value occurs.

`distinct`	`()`
	`(partitions: Int)`
		It returns an RDD that contains only distinct elements.

`filter`	`(f: T => Boolean)`	It returns an RDD that contains only elements matching `f`.
`filterWith`	`(constructA: Int => A )(f: (T, A) => Boolean)`	It is similar to filter, but `f` takes an additional parameter generated by `constructA`, which is called per-partition. The original motivation for this came from providing PRNG generation per shard.
`first`	`()`	It returns the "first" element of the RDD.
`flatMap`	`(f: T => TraversableOnce[U])`	It returns an RDD of type `U`.
`fold`	`(zeroValue: T)(op: (T,T) => T)`	It merges values using the provided operation, first on each partition, and then merges the merged result.
`foreach`	`(f: T => Unit)`	It applies the function `f` to each element.

`groupBy`	`(f: T => K)`
	`(f: T => K, p: Partitioner)`
	`(f: T => K, numPartitions:Int)`
		It takes in an RDD and produces a PairRDD of type (K,Seq[T]) using the result of `f` for the key for each element.

Function	**Arguments**	**Returns**

`keyBy`	`(f: T => K)`
	`(f: T => K, p: Partitioner)`
	`(f: T => K, numPartitions:Int)`
		It is the same as `groupBy` but does not group results together with duplicate keys. It returns an RDD of (K,T).

`map`	`(f: T => U)`	It returns an RDD of the result of applying `f` to every element in the input RDD.
`mapPartitions`	`(f: Iterator[T] => Iterator[U])`	It is similar to `map` except that the provided function takes and returns an iterator and is applied to each partition.
`mapPartitionsWithIndex`	`(f: (Int, Iterator[T]) => Iterator[U], preservePartitions)`	It is the same as `mapPartitions` but also provides the index of the original partition.
`mapWith`	`(constructA: Int => A)(f: (T, A) => U)`	It is similar to `map`, but `f` takes an additional parameter generated by `constructA`, which is called per-partition. The original motivation for this came from providing PRNG generation per shard.

`persist`	`()`
	`(newLevel: StorageLevel)`
		Sets the RDD storage level, which can cause the RDD to be stored after it is computed. Different `StorageLevel` values can be seen in `StorageLevel.scala` (`NONE`, `DISK_ONLY`, `MEMORY_ONLY`, and `MEMORY_AND_DISK` are the common ones).

`pipe`	`(command: Seq[String])`
	`(command: Seq[String], env: Map[String, String])`
		It takes an RDD and calls the specified command with the optional environment. Then, it pipes each element through the command. That results in an RDD of type string.

`sample`	`(withReplacement: Boolean, fraction: Double, seed: Int)`	It returns an RDD of that fraction.
`takeSample`	`(withReplacement: Boolean, num: Int, seed: Int)`	It returns an array of the requested number of elements. It works by over sampling the RDD and then grabbing a subset.
`toDebugString`	`()`	It's a handy function that outputs the recursive deps of the RDD.

Function	Arguments	Returns
`union`	`(other: RDD[T])`	It's an RDD containing elements of both RDDs. Here, duplicates are not removed.
`unpersist`	`()`	Remove all blocks of the RDD from memory/disk if they've persisted.
`zip`	`(other: RDD[U])`	It is important to note that it requires that the RDDs have the same number of partitions and the same size of each partition. It returns an RDD of key-value pairs RDD[T,U].

Java RDD functions

Many of the Java RDD functions are quite similar to the Scala RDD functions, but the type signatures are somewhat different.

Spark Java function classes

For the Java RDD API, we need to extend one of the provided function classes while implementing our function:

Name	Params	Purpose
`Function<T,R>`	`R call(T t)`	It is a function that takes something of type `T` and returns something of type `R`. It is commonly used for maps.
`DoubleFunction<T>`	`Double call(T t)`	It is the same as `Function<T, Double>`, but the result of the map-like call returns a JavaDoubleRDD (for summary statistics).
`PairFunction<T, K, V>`	`Tuple2<K, V> call(T t)`	It is a function that results in a JavaPairRDD. If you're working on `JavaPairRDD<A,B>`, have `T` of type `Tuple2<A,B>`.
`FlatMap Function<T, R>`	`Iterable<R> call(T t)`	It is a function for producing a RDD through `flatMap`.
`PairFlatMap Function<T, K, V>`	`Iterable<Tuple2<K, V>> call(T t)`	It's a function that results in a JavaPairRDD. If you're working on `JavaPairRDD<A,B>`, have `T` of type `Tuple2<A,B>`.

Name	Params	Purpose
`DoubleFlatMap Function<T>`	`Iterable<Double> call(T t)`	It is the same as `FlatMapFunction<T, Double>`, but the result of the map-like call returns a JavaDoubleRDD (for summary statistics).
`Function2<T1, T2, R>`	`R call(T1 t1, T2 t2)`	It is a function for taking two inputs and returning an output. It is used by fold and similar.

Common Java RDD functions

These RDD functions are available regardless of the type of RDD.

Name	Params	Purpose
`cache`	`()`	It makes an RDD persist in memory.
`coalesce`	`numPartitions: Int`	It returns a new RDD with `numPartitions` partitions.
`collect`	`()`	It returns the List representation of the entire RDD.
`count`	`()`	It returns the number of elements.
`countByValue`	`()`	It returns a map of each unique value to the number of times that value shows up.

`distinct`	`()`
	`(Int numPartitions)`
		It is an RDD consisting of all of the distinct elements of the RDD, optionally in the provided number of partitions.

`filter`	`(Function<T, Boolean> f)`	It is an RDD consisting only of the elements for which the provided function returns `true`.
`first`	`()`	It is the first element of the RDD.

`flatMap`	`(FlatMapFunction<T, U> f)`
	`(DoubleFlatMapFunction<T> f)`
	`(PairFlatMapFunction<T, K, V> f)`
		It is an RDD of the specified types (`U`, `Double` and `Pair<K,V>` respectively).

`fold`	`(T zeroValue, Function2<T, T, T> f)`	It returns the result `T`. Each partition is folded individually with the zero value and then the results are folded.

Name	Params	Purpose
`foreach`	`(VoidFunction<T> f)`	It applies the function to each element in the RDD.
`groupBy`	`(Function<T, K> f)` `(Function<T, K> f, Int numPartitions)`	It returns a JavaPairRDD of grouped elements.
`map`	`(DoubleFunction<T> f)` `(PairFunction<T, K2, V2> f)` `(Function<T, U> f)`	It returns an RDD of an appropriate type for the input function (see previous table) by calling the provided function on each element in the input RDD.
`mapPartitions`	`(DoubleFunction <Iterator<T>> f)` `(PairFunction <Iterator<T>, K2, V2> f)` `(Function<Iterator<T>, U> f)`	It is similar to map, but the provided function is called per-partition. This can be useful if you have done some setup work that is necessary for each partition.
`reduce`	`(Function2<T, T, T> f)`	It uses the provided function to reduce down all of the elements.
`sample`	`(Boolean withReplacement, Double fraction, Int seed)`	It returns a smaller RDD consisting of only the requested fraction of the data.

Methods for combining JavaRDDs

There are a number of different functions that we can use to combine RDDs:

Name	Params	Purpose
`subtract`	`(JavaRDD<T> other)` `(JavaRDD<T> other, Partitioner p)` `(JavaRDD<T> other, Int numPartitions)`	It returns an RDD with only the elements initially present in the first RDD and not present in the other RDD.
`union`	`(JavaRDD<T> other)`	It is the union of the two RDDs.
`zip`	`(JavaRDD<U> other)`	It returns an RDD of key-value pairs RDD[T,U]. It is important to note that it requires that the RDDs should have the same number of partitions and the size of each partition.

Functions on JavaPairRDDs

Some functions are only defined on key-value PairRDDs:

Name	Params	Purpose

`cogroup`	`(JavaPairRDD<K, W> other)`
	`(JavaPairRDD<K, W> other, Int numPartitions)`
	`(JavaPairRDD<K, W> other1, JavaPairRDD<K, W> other2)`
	`(JavaPairRDD<K, W> other1, JavaPairRDD<K, W> other2, Int numPartitions)`
		It joins two (or more) RDDs by the shared key. Note that if an element is missing in one RDD but present in the other one, the list will simply be empty.

`combineByKey`	`(Function<V, C> createCombiner`
	`Function2<C, V, C> mergeValue,`
	`Function2<C,C,C> mergeCombiners)`
		It's a generic function to combine elements by key. The `createCombiner` function turns something of type `V` into something of type `C`. The `mergeValue` function adds `V` to `C` and `mergeCombiners` is used to combine two `C` values into a single `C` value.

`collectAsMap`	`()`	It returns a map of the key-value pairs.
`countByKey`	`()`	It returns a map of the key to the number of elements with that key.
`flatMapValues`	`(Function[T] f, Iterable[V] v)`	It returns an RDD of type `V`.

`join`	`(JavaPairRDD<K, W> other)`
	`(JavaPairRDD<K, W> other, Int integers)`
		It joins an RDD with another RDD. The result is only present for elements where the key is present in both the RDDs.

`keys`	`()`	It returns an RDD of only the keys.
`lookup`	`(Key k)`	It looks up a specific element in the RDD. It uses the RDD's partitioner to figure out which shard(s) to look at.

Name	Params	Purpose
`reduceByKey`	`(Function2[V,V,V] f)`	The `reduceByKey` function is the parallel version of `reduce` that merges the values for each key using the provided function and returns an RDD.

`sortByKey`	`(Comparator[K] comp, Boolean ascending)`
	`(Comparator[K] comp)`
	`(Boolean ascending)`
		It sorts the RDD by key; so each partition contains a fixed range.

`values`	`()`	It returns an RDD of only the values.

Manipulating your RDD in Python

Spark has a more limited Python API than Java and Scala, but it supports for most of the core functionality.

The hallmark of a MapReduce system are the two commands `map` and `reduce`. You've seen the `map` function used in the past chapters. The `map` function works by taking in a function that works on each individual element in the input RDD and produces a new output element. For example, to produce a new RDD where you have added one to every number, you would use `rdd.map(lambda x: x+1)`. It's important to understand that the `map` function and the other Spark functions, do not transform the existing elements; rather they return a new RDD with new elements. The `reduce` function takes a function that operates on pairs to combine all the data. This is returned to the calling program. If you were to sum all of the elements, you would use `rdd.reduce(lambda x, y: x+y)`. The `flatMap` function is a useful utility function that allows you to write a function that returns an iterable of the type you want and then flattens the results. A simple example of this is a case where you want to parse all of the data, but some of it might fail to parse. The `flatMap` function can output an empty list if it has failed or a list with its success if it has worked. In addition to `reduce`, there is a corresponding `reduceByKey` function that works on RDDs, which are key-value pairs, and produces another RDD.

Many of the mapping operations are also defined with a partition's variant. In this case, the function you need to provide takes and returns an iterator, which represents all of the data on that partition, thus performing work on a per-partition level. The `mapPartitions(func)` function can be quite useful if the operation you need to perform has to do expensive work on each shard/partition. An example of this is establishing a connection to a backend server. Another reason for using `mapPartitions(func)` is to do setup work for your `map` function that can't be serialized across the network. A good example of this is parsing some expensive side input, as shown here:

```
def f(iterator):
      // Expensive work goes here
     for i in iterator:
          yield per_element_function(i)
```

Often, your data can be expressed with key-value mappings. As such, many of the functions defined on Python's RDD class only work if your data is in a key-value mapping. The `mapValues` function is used when you only want to update the key-value pair you are working with.

In addition to performing simple operations on the data, Spark also provides support for broadcast values and accumulators. Broadcast values can be used to broadcast a read-only value to all of the partitions, which can save the need to re-serialize a given value multiple times. Accumulators allow all of the shards to add to the accumulator and the result can then be read on the master. You can create an accumulator by doing `counter = sc.accumulator(initialValue)`. If you want customized add behavior, you can also provide an `AccumulatorParam` to the accumulator. The return can then be incremented as `counter += x` on any of the workers. The resulting value can then be read with `counter.value()`. The broadcast value is created with `bc = sc.broadcast(value)` and then accessed by `bc.value()` on any worker. The accumulator can only be read on the master, and the broadcast value can be read on all of the shards.

Let's look at a quick Python example that shows multiple RDD operations. We have two text files `2009-2014-BO.txt` and `1861-1864-AL.txt`. These are the *State Of the Union* speeches by Presidents Barack Obama and Abraham Lincoln. We want to compare the mood of the nation by comparing the salient difference in the words used.

The first step is reading the files and creating the word frequency vector, that is, each word and the number of times it is used in the speech. I am sure you would recognize this as a canonical word count MapReduce example and, in traditional Hadoop Map Reduce, it takes around 100 lines of code. In Spark, as we shall see, it takes only 5 lines of code:

```
from pyspark.context import SparkContext
print "Running Spark Version %s" % (sc.version)
```

```
from pyspark.conf import SparkConf
conf = SparkConf()
print conf.toDebugString()
```

The MapReduce code is shown here:

```
from operator import add
lines = sc.textFile("sotu/2009-2014-BO.txt")
word_count_bo = lines.flatMap(lambda x: x.split(' ')).\
    map(lambda x: (x.lower().rstrip().
      lstrip().rstrip(',').rstrip('.'), 1)).\
    reduceByKey(add)
word_count_bo.count()
#6658 without lower, 6299 with lower, rstrip,lstrip 4835
lines = sc.textFile("sotu/1861-1864-AL.txt")
word_count_al = lines.flatMap(lambda x: x.split(' ')).map(lambda
  x: (x.lower().rstrip().lstrip().rstrip(',').rstrip('.'),
  1)).reduceByKey(add)
word_count_al.count()
```

Sorting an RDD by any column is very easy as shown next:

```
word_count_bo_1 = word_count_bo.sortBy(lambda x:
  x[1],ascending=False)
```

We can collect the word vector. But don't print it! It is a long list:

```
for x in word_count_bo_1.take(10):
    print x
```

Now, let's take out common words, as shown here:

```
common_words = ["us","has","all", "they", "from", "who","what","on",
"by","more","as","not","their","can","new","it","but","be","are","--
","i","have","this","will","for","with","is","that","in","our","we","
a","of","to","and","the","that's","or","make","do","you","at","it\'s"
,"than","if","know","last","about","no","just","now","an","because","
<p>we","why","we\'ll","how","two","also","every","come","we've","year"
,"over","get","take","one","them","we\'re","need","want","when","like"
,"most","-","been","first","where","so","these","they\'re","good","wou
ld","there","should","-->","<!--","up","i\'m","his","their","which","m
ay","were","such","some","those","was","here","she","he","its","her","
his","don\'t","i\'ve","what\'s","didn\'t","shouldn\'t","(applause.)","
let\'s","doesn\'t"]
```

Filtering out common words is also a single filter operation. Of course, as RDDs are immutable, we would create a new filtered RDD:

```
word_count_bo_clean = word_count_bo_1.filter(lambda x: x[0] not in
  common_words)
word_count_al_clean = word_count_al.filter(lambda x: x[0] not in
  common_words)
```

Finding the words that were spoken by Obama but not by Lincoln, is a single RDD operation. You need to use `subractByKey` and then use `sortBy` on the count to see the different but most frequent words, as shown here:

```
for x in word_count_bo_clean.subtractByKey
  (word_count_al_clean).sortBy(lambda x:
  x[1],ascending=False).take(15): #collect():
    print x
```

The preceding program should give you a good grip on the RDD functions and how to use them in Python.

Standard RDD functions

These functions are available on all RDDs in Python:

Name	Params	Purpose
`flatMap`	`f, preserves Partitioning=False`	It takes a function that returns an iterator of type `U` for each input of type `T` and returns a flattened RDD of type `U`.
`mapParitions`	`f, preserves Partitioning=False`	It takes a function that takes in an iterator of type `T` and returns an iterator of type `U`, which then results in an RDD of type `U`. It's useful for map operations with expensive per machine setup work.
`filter`	`f`	It takes a function and returns an RDD with only the elements for which the function returns `true`.
`distinct`	`()`	It returns an RDD with distinct elements (for example, 1, 1, 2 gives the output as 1, 2).
`union`	other	It returns a union of two RDDs.
`cartesian`	other	It returns the cartesian product of the RDD with the other RDD.

Name	Params	Purpose
`groupBy`	`f, numPartitions=None`	It returns an RDD with the elements grouped together for the value that `f` outputs.
`pipe`	`command, env={}`	It pipes each element of the RDD to the provided command and returns an RDD of the result.
`foreach`	`f`	It applies the function `f` to each element in the RDD.
`reduce`	`f`	It reduces the elements using the provided function.
`fold`	`zeroValue, op`	Each partition is folded individually with zero value and then the results are folded.
`countByValue`	`()`	It returns a dictionary mapping of each distinct value to the number of times it is found in the RDD.
`take`	`num`	It returns a list of `num` elements. This can be slow for large values of `num`; so use `collect` if you want to get back the entire RDD.
`partitionBy`	`numPartitions, partitionFunc=hash`	Make a new RDD partitioned by the provided partitioning function. The `partitionFunc` function simply needs to map the input key to an integer number and the `partitionBy` calculates the partition by that number mod `numPartitions`.

PairRDD functions

These functions are only available on key-value pair functions:

Name	Params	Purpose
`collectAsMap`	`()`	This returns a dictionary consisting of all of the key-value pairs of the RDD.
`reduceByKey`	`func, numPartitions=None`	The `reduceByKey` function is the parallel version of `reduce`, which merges the values for each key using the provided function and returns an RDD.
`countByKey`	`()`	This returns a dictionary of the number of elements for each key.

Name	**Params**	**Purpose**
`join`	`other, numPartitions=None`	This joins an RDD with another RDD. The result is only present for elements where the key is present in both RDDs. The value that gets stored for each key is a tuple of the values from each RDD.
`rightOuterJoin`	`other, numPartitions=None`	This joins an RDD with another RDD. It outputs a given key-value pair only if the key it's being joined with is present in the RDD. If key is not present in the source RDD, the first value in the tuple will be `None`.
`leftOuterJoin`	`other, numPartitions=None`	This joins an RDD with another RDD. It outputs a given key-value pair only if the key is present in the source RDD. If the key is not present in other RDD, the second value in the tuple will be `None`.
`combineByKey`	`createCombiner, mergeValues, mergeCombiners`	This combines elements by key. It takes an RDD of type (K,V) and returns an RDD of type (K,C). The `createCombiner` function turns something of type `V` into something of type `C`. The `mergeValue` function adds a `V` to a `C`, and `mergeCombiners` is used to combine two `C` values into a single `C` value.
`zip`	`other`	This returns key-value pairs, pairing one element from each RDD. The first key-value pair would be the 1st element from this RDD, and the value would be the 1st element from the "other" RDD; the second pair would be the respective second elements from each of the RDDs and so on.
`groupByKey`	`numPartitions=None`	This groups the values in the RDD by the key they have.
`cogroup`	`other, numPartitions=None`	This joins two (or more) RDDs by the shared key. Note that if an element is missing in one RDD but present in the other one, the list will simply be empty.

Some references are as follows:

- `http://www.scala-lang.org/api/current/index.html#scala.collection.immutable.List`
- `http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.api.java.JavaRDD`
- `http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.api.java.JavaPairRDD`
- `http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.api.java.JavaDoubleRDD`
- `https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.SparkContext`
- `http://abshinn.github.io/python/apache-spark/2014/10/11/using-combinebykey-in-apache-spark/`
- Good examples of RDD transformations (`https://github.com/JerryLead/SparkLearning/tree/master/src`)

Summary

This chapter looked at how to perform computations on data in a distributed fashion once it's loaded into an RDD. With our knowledge of how to load and save RDDs, we can now write distributed programs using Spark.

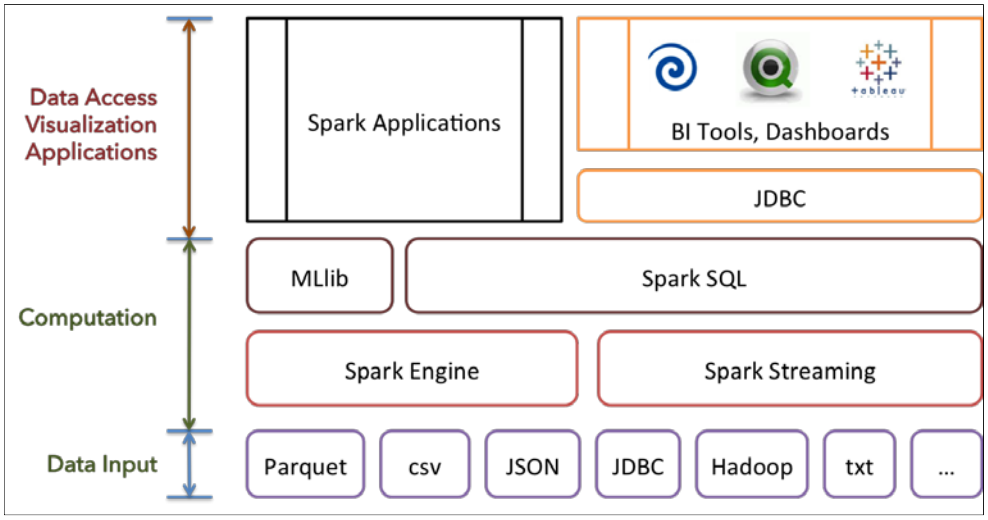
7
Spark SQL

Spark SQL holds an important feature in the Spark ecosystem, that is, integration with different data sources as well as the capability to interact with other subsystems such as visualization. As we know that in modern data stacks, no stack is an island by itself and in many ways, the versatility of integration with other components is an important capability. Obviously, the role of Spark SQL is not to replace SQL databases. We see it more as a versatile query interface to Spark data that complements the data wrangling and input capabilities of Spark. The ability to scale complex data operations makes sense only when one can utilize the results in flexible ways and Spark SQL achieves that. We'll cover the following topics in this chapter:

- Interfacing Spark to dashboards (such as Tableau and Qlik) that know how to fire off SQL statements from a visualization interface based on what a user selects.
- Another use case for Spark SQL is programming queries to Spark data without employing RDD semantics. While RDD manipulations are required to implement data algorithms, the final dataset can be in a SchemaRDD, which can be queried using SQL. Sometimes, a combination of both works very well.
- Leveraging the knowledge of SQL queries. There is a huge amount of SQL knowledge among various people with roles ranging from data analysts and programmers to data engineers who have developed interesting SQL queries over their data. Spark needs to leverage that and it does that via Spark SQL.

The Spark SQL architecture

Interestingly as I was writing this chapter, Michael Armbrust from Databricks wrote a blog about the data sources API and an architecture diagram, from which I got the inspiration to create the following diagram:



The bottom layer is the flexible data access (and store) that works via multiple formats, usually a distributed filesystem such as the HDFS. The computation layer is the place where we leverage the distributed-at-scale processing of the Spark engine including the streaming data. The computation layer usually acts on RDDs. The Spark SQL then overlays the SchemaRDD veneer and provides the data access for applications, dashboards, BI tools, and so forth.

Spark SQL how-to in a nutshell

The heart of the Spark SQL is the SchemaRDD, which, as you can guess, associates a schema with an RDD. Of course, internally it does a lot of magic by leveraging the ability to scale and distribute processing, and that of flexible storage.

In many ways, the data access via Spark SQL is deceptively simple, that is, creating one or more appropriate RDDs paying attention to the layout, data types, and so on and then accessing via SchemaRDDs. We get to use all the interesting features of Spark for creating the RDDs: structured data from Hive or Parquet, unstructured data from any source, and the ability to apply the RDD operations at scale. Then you need to overlay respective schemas to the RDDs by creating SchemaRDDs. Viola! You now have the ability to run SQL over RDDs. You can see the SchemaRDDs being created in the log entries.

Spark SQL programming

Let's not get our hands dirty and work through various examples. We will start with a simple dataset and then progressively perform more sophisticated SQL statements. While writing other chapters, I was wondering what a good dataset that brings out the various aspects of SQL would be. And I hit upon an idea! Long time ago, the Northwind database was the canonical database to learn Microsoft Access and later SQL server. And that would be a good dataset for learning Spark SQL as well!

Let's use some of the tables and data to dig deeper into Spark SQL. The SQL scripts to create the Northwind database is available at `https://northwinddatabase.codeplex.com/releases/view/71634`. In our case, we will load data from a set of CSV files and create an appropriate SchemaRDDs in Spark. Then we will fire off SQL queries of increasing complexity. A good reference for this is the Spark SQL programming guide available at `https://spark.apache.org/docs/latest/sql-programming-guide.html`.

SQL access to a simple data table

Let's load a small CSV file to the `employee` table, as shown here:

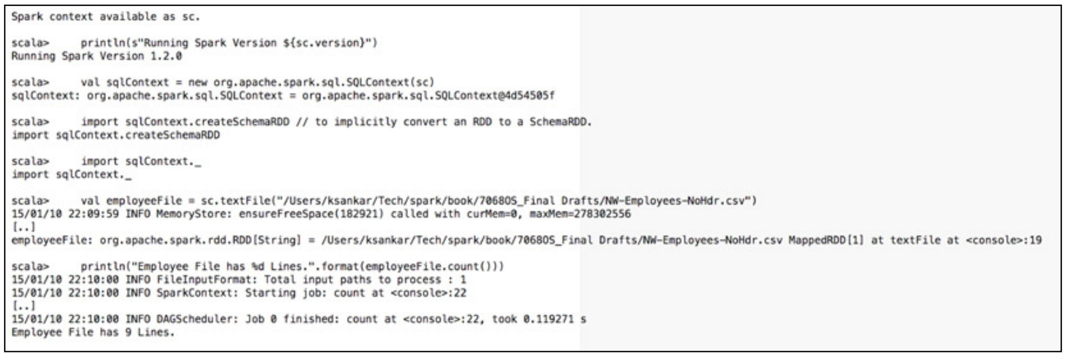
```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._ // for implicit conversations
import org.apache.spark.sql._

object BigData01 {
  // register case class external to main
  case class Employee(EmployeeID : Int,
    LastName : String, FirstName : String, Title : String,
    BirthDate : String, HireDate : String,
    City : String, State : String, Zip : String, Country : String,
    ReportsTo : String)
    //
  def main(args: Array[String]): Unit = {
val sc = new SparkContext("local","Chapter 7")
    println(s"Running Spark Version ${sc.version}")
    //
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
import sqlContext.createSchemaRDD // to implicitly convert an RDD
  to a SchemaRDD.
import sqlContext._
    //
val employeeFile = sc.textFile("/Users/ksankar/fdps-vii/NW-
  Employees-NoHdr.csv")
```

```
      println("Employee File has %d Lines."
        .format(employeeFile.count()))
      val employees = employeeFile.map(_.split(",")).
        map(e => Employee( e(0).trim.toInt,
          e(1), e(2), e(3),
          e(4), e(5),
          e(6), e(7), e(8), e(9), e(10)))
       println(employees.count)
       employees.registerTempTable("Employees")
       var result = sqlContext.sql("SELECT * from Employees")
       result.foreach(println)
       result = sqlContext.sql("SELECT * from Employees WHERE State =
'WA'")
       result.foreach(println)
    }
  }
```

The code is straightforward. We create a `case` class that represents the `employee` table. We then parse the CSV file and create an RDD that has the `Employee` classes as its elements.

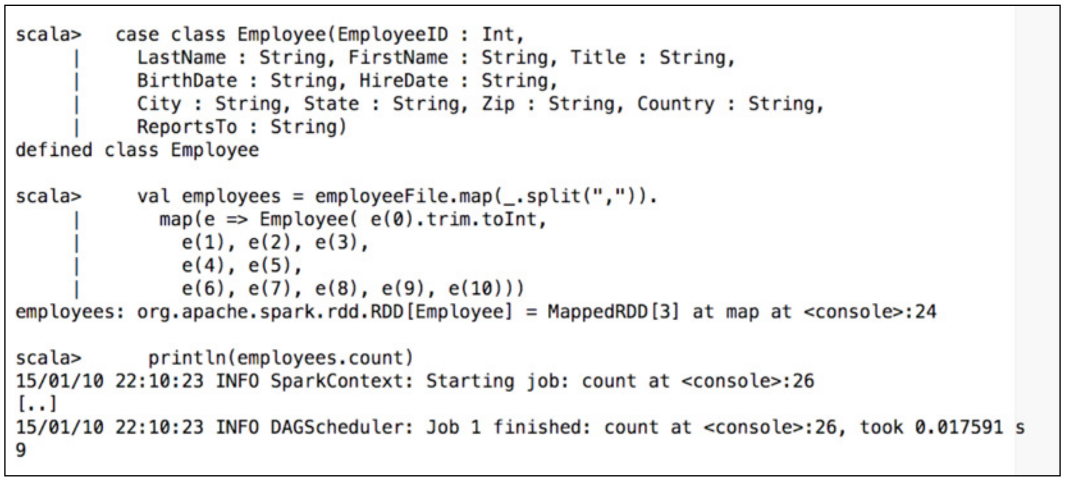
The datafiles are available from at `https://github.com/xsankar/fdps-vii`.

The screenshot of the process and output of running the code from the Spark shell is shown here:



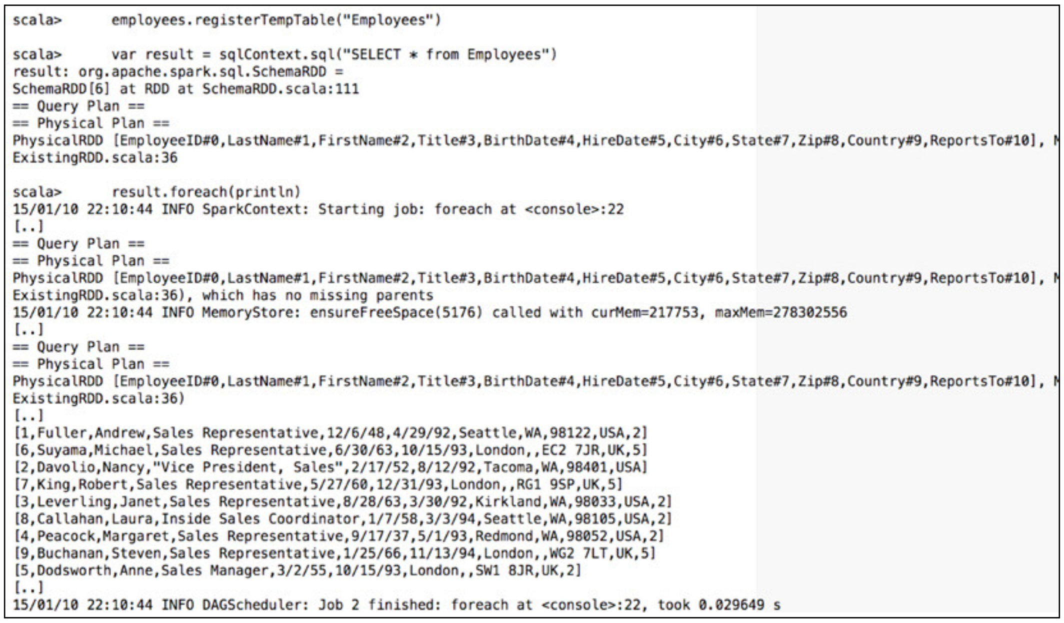
```
Spark context available as sc.

scala>     println(s"Running Spark Version ${sc.version}")
Running Spark Version 1.2.0

scala>     val sqlContext = new org.apache.spark.sql.SQLContext(sc)
sqlContext: org.apache.spark.sql.SQLContext = org.apache.spark.sql.SQLContext@4d54505f

scala>     import sqlContext.createSchemaRDD // to implicitly convert an RDD to a SchemaRDD.
import sqlContext.createSchemaRDD

scala>     import sqlContext._
import sqlContext._

scala>     val employeeFile = sc.textFile("/Users/ksankar/Tech/spark/book/70680S_Final Drafts/NW-Employees-NoHdr.csv")
15/01/10 22:09:59 INFO MemoryStore: ensureFreeSpace(182921) called with curMem=0, maxMem=278302556
[..]
employeeFile: org.apache.spark.rdd.RDD[String] = /Users/ksankar/Tech/spark/book/70680S_Final Drafts/NW-Employees-NoHdr.csv MappedRDD[1] at textFile at <console>:19

scala>     println("Employee File has %d Lines.".format(employeeFile.count()))
15/01/10 22:10:00 INFO FileInputFormat: Total input paths to process : 1
15/01/10 22:10:00 INFO SparkContext: Starting job: count at <console>:22
[..]
15/01/10 22:10:00 INFO DAGScheduler: Job 0 finished: count at <console>:22, took 0.119271 s
Employee File has 9 Lines.
```

We declare a `case` class and parse the file to `RDD[Employee]`, as shown here:



```
scala>   case class Employee(EmployeeID : Int,
     |     LastName : String, FirstName : String, Title : String,
     |     BirthDate : String, HireDate : String,
     |     City : String, State : String, Zip : String, Country : String,
     |     ReportsTo : String)
defined class Employee

scala>     val employees = employeeFile.map(_.split(",")).
     |       map(e => Employee( e(0).trim.toInt,
     |         e(1), e(2), e(3),
     |         e(4), e(5),
     |         e(6), e(7), e(8), e(9), e(10)))
employees: org.apache.spark.rdd.RDD[Employee] = MappedRDD[3] at map at <console>:24

scala>      println(employees.count)
15/01/10 22:10:23 INFO SparkContext: Starting job: count at <console>:26
[..]
15/01/10 22:10:23 INFO DAGScheduler: Job 1 finished: count at <console>:26, took 0.017591 s
9
```

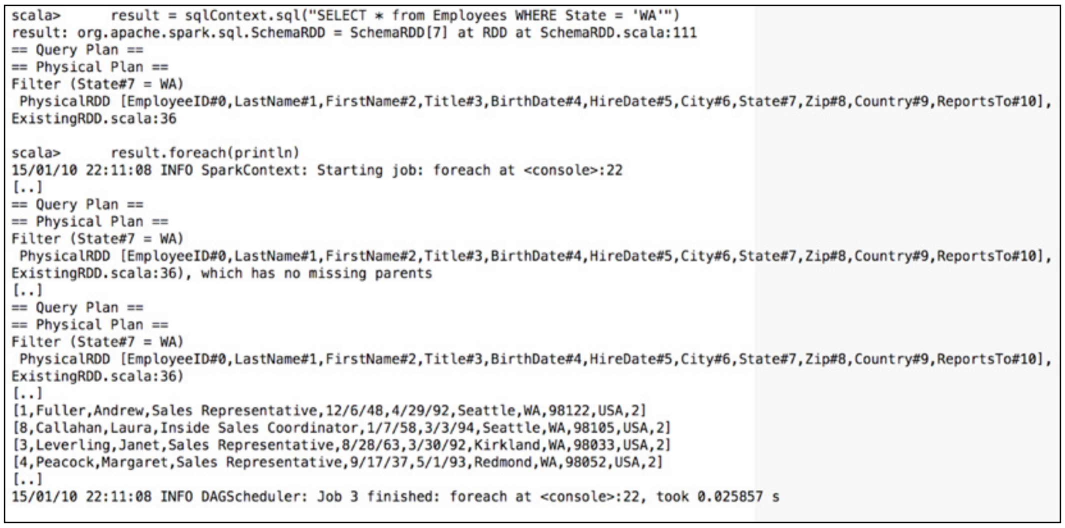
Now, you'll learn about the SQL magic. We turn the RDD into a SchemaRDD and then run SQL queries, as shown in this screenshot:



```
scala>      employees.registerTempTable("Employees")

scala>      var result = sqlContext.sql("SELECT * from Employees")
result: org.apache.spark.sql.SchemaRDD =
SchemaRDD[6] at RDD at SchemaRDD.scala:111
== Query Plan ==
== Physical Plan ==
PhysicalRDD [EmployeeID#0,LastName#1,FirstName#2,Title#3,BirthDate#4,HireDate#5,City#6,State#7,Zip#8,Country#9,ReportsTo#10],
ExistingRDD.scala:36

scala>      result.foreach(println)
15/01/10 22:10:44 INFO SparkContext: Starting job: foreach at <console>:22
[..]
== Query Plan ==
== Physical Plan ==
PhysicalRDD [EmployeeID#0,LastName#1,FirstName#2,Title#3,BirthDate#4,HireDate#5,City#6,State#7,Zip#8,Country#9,ReportsTo#10],
ExistingRDD.scala:36), which has no missing parents
15/01/10 22:10:44 INFO MemoryStore: ensureFreeSpace(5176) called with curMem=217753, maxMem=278302556
[..]
== Query Plan ==
== Physical Plan ==
PhysicalRDD [EmployeeID#0,LastName#1,FirstName#2,Title#3,BirthDate#4,HireDate#5,City#6,State#7,Zip#8,Country#9,ReportsTo#10],
ExistingRDD.scala:36)
[..]
[1,Fuller,Andrew,Sales Representative,12/6/48,4/29/92,Seattle,WA,98122,USA,2]
[6,Suyama,Michael,Sales Representative,6/30/63,10/15/93,London,,EC2 7JR,UK,5]
[2,Davolio,Nancy,"Vice President, Sales",2/17/52,8/12/92,Tacoma,WA,98401,USA]
[7,King,Robert,Sales Representative,5/27/60,12/31/93,London,,RG1 9SP,UK,5]
[3,Leverling,Janet,Sales Representative,8/28/63,3/30/92,Kirkland,WA,98033,USA,2]
[8,Callahan,Laura,Inside Sales Coordinator,1/7/58,3/3/94,Seattle,WA,98105,USA,2]
[4,Peacock,Margaret,Sales Representative,9/17/37,5/1/93,Redmond,WA,98052,USA,2]
[9,Buchanan,Steven,Sales Representative,1/25/66,11/13/94,London,,WG2 7LT,UK,5]
[5,Dodsworth,Anne,Sales Manager,3/2/55,10/15/93,London,,SW1 8JR,UK,2]
[..]
15/01/10 22:10:44 INFO DAGScheduler: Job 2 finished: foreach at <console>:22, took 0.029649 s
```

You can see the query plan and see that finally an RDD is returned as the query result.

Let us try a filter query `SELECT * from Employees WHERE State = 'WA'` and see how it works. Here is a screenshot of this:



```
scala>      result = sqlContext.sql("SELECT * from Employees WHERE State = 'WA'")
result: org.apache.spark.sql.SchemaRDD = SchemaRDD[7] at RDD at SchemaRDD.scala:111
== Query Plan ==
== Physical Plan ==
Filter (State#7 = WA)
 PhysicalRDD [EmployeeID#0,LastName#1,FirstName#2,Title#3,BirthDate#4,HireDate#5,City#6,State#7,Zip#8,Country#9,ReportsTo#10],
ExistingRDD.scala:36

scala>      result.foreach(println)
15/01/10 22:11:08 INFO SparkContext: Starting job: foreach at <console>:22
[..]
== Query Plan ==
== Physical Plan ==
Filter (State#7 = WA)
 PhysicalRDD [EmployeeID#0,LastName#1,FirstName#2,Title#3,BirthDate#4,HireDate#5,City#6,State#7,Zip#8,Country#9,ReportsTo#10],
ExistingRDD.scala:36), which has no missing parents
[..]
== Query Plan ==
== Physical Plan ==
Filter (State#7 = WA)
 PhysicalRDD [EmployeeID#0,LastName#1,FirstName#2,Title#3,BirthDate#4,HireDate#5,City#6,State#7,Zip#8,Country#9,ReportsTo#10],
ExistingRDD.scala:36)
[..]
[1,Fuller,Andrew,Sales Representative,12/6/48,4/29/92,Seattle,WA,98122,USA,2]
[8,Callahan,Laura,Inside Sales Coordinator,1/7/58,3/3/94,Seattle,WA,98105,USA,2]
[3,Leverling,Janet,Sales Representative,8/28/63,3/30/92,Kirkland,WA,98033,USA,2]
[4,Peacock,Margaret,Sales Representative,9/17/37,5/1/93,Redmond,WA,98052,USA,2]
[..]
15/01/10 22:11:08 INFO DAGScheduler: Job 3 finished: foreach at <console>:22, took 0.025857 s
```

Great, it worked as expected! You can see that the filter did get into the query plan.

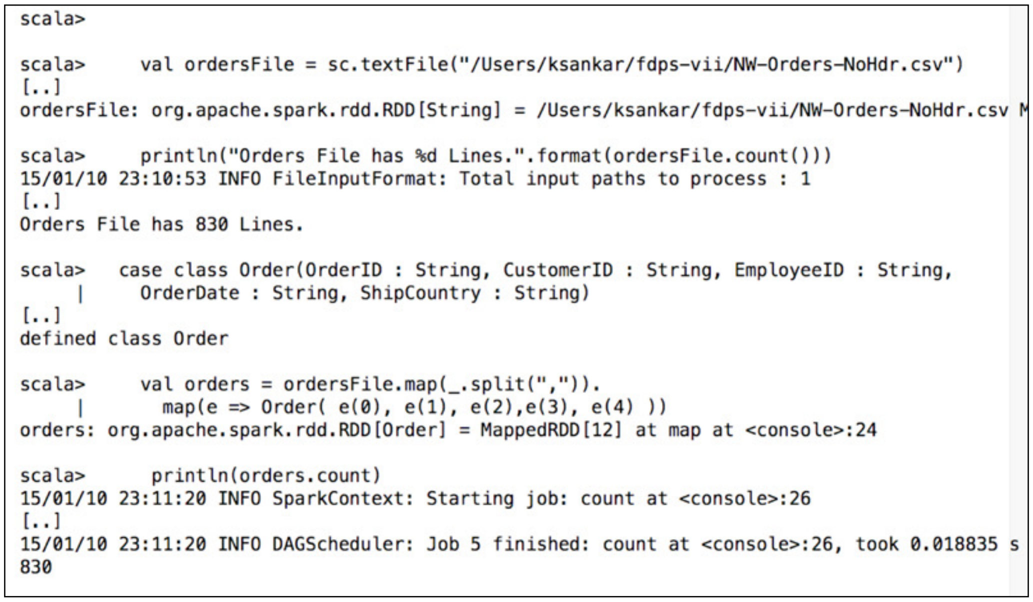
Handling multiple tables with Spark SQL

Now that we have mastered the art of Spark SQL, let's try multiple datasets and slightly larger datasets. The `Orders` table's dataset has 830 records and the `Order Details` has approximately 2000 records. These would give us a good representation of a few queries with joins that span the two tables.

Let's start by loading the `Orders` table, as shown next:

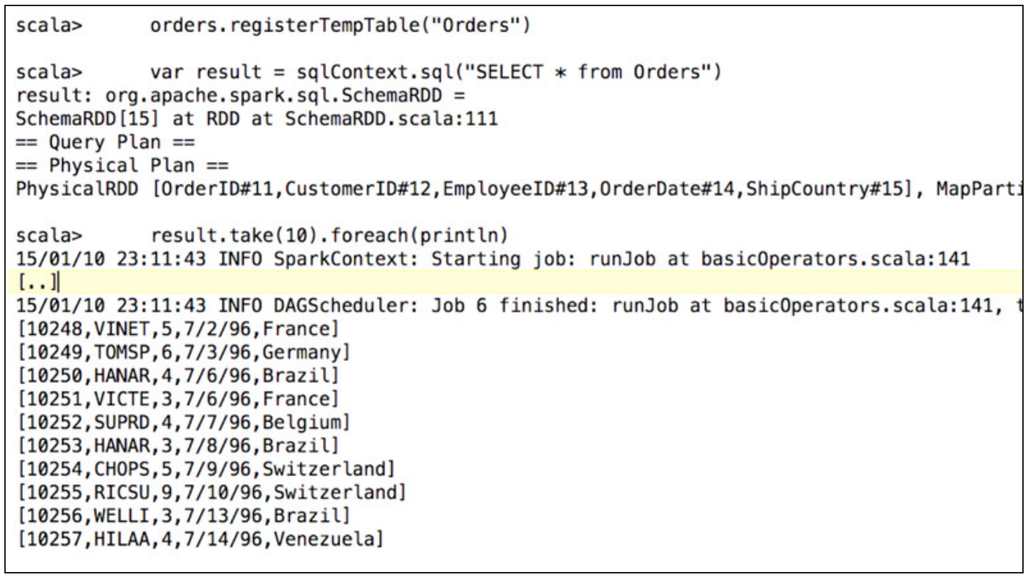
```
val ordersFile = sc.textFile("/Users/ksankar/fdps-vii/NW-Orders-
  NoHdr.csv")
    println("Orders File has %d Lines."
      .format(ordersFile.count()))
    val orders = ordersFile.map(_.split(",")).
      map(e => Order( e(0), e(1), e(2),e(3), e(4) ))
     println(orders.count)
     orders.registerTempTable("Orders")
     var result = sqlContext.sql("SELECT * from Orders")
     result.take(10).foreach(println)
     //
```

The output of this is shown in the next screenshot. This is nothing different from our earlier work. You can see where it casts the variable result as a SchemaRDD. We have 830 orders in our table, as you can see here:



```
scala>

scala>     val ordersFile = sc.textFile("/Users/ksankar/fdps-vii/NW-Orders-NoHdr.csv")
[..]
ordersFile: org.apache.spark.rdd.RDD[String] = /Users/ksankar/fdps-vii/NW-Orders-NoHdr.csv M

scala>     println("Orders File has %d Lines.".format(ordersFile.count()))
15/01/10 23:10:53 INFO FileInputFormat: Total input paths to process : 1
[..]
Orders File has 830 Lines.

scala>   case class Order(OrderID : String, CustomerID : String, EmployeeID : String,
     |     OrderDate : String, ShipCountry : String)
[..]
defined class Order

scala>     val orders = ordersFile.map(_.split(",")).
     |       map(e => Order( e(0), e(1), e(2),e(3), e(4) ))
orders: org.apache.spark.rdd.RDD[Order] = MappedRDD[12] at map at <console>:24

scala>      println(orders.count)
15/01/10 23:11:20 INFO SparkContext: Starting job: count at <console>:26
[..]
15/01/10 23:11:20 INFO DAGScheduler: Job 5 finished: count at <console>:26, took 0.018835 s
830
```

In this chapter, we are trying to create a few queries. So we really do not need hundreds of records. But the dataset has more records so that you can try out various queries on your own. The dataset is big enough to do meaningful queries but small enough to work on a laptop with limited resources. This would be a good exercise for you to experiment with Spark SQL. Look at the following screenshot for the results of this exercise:

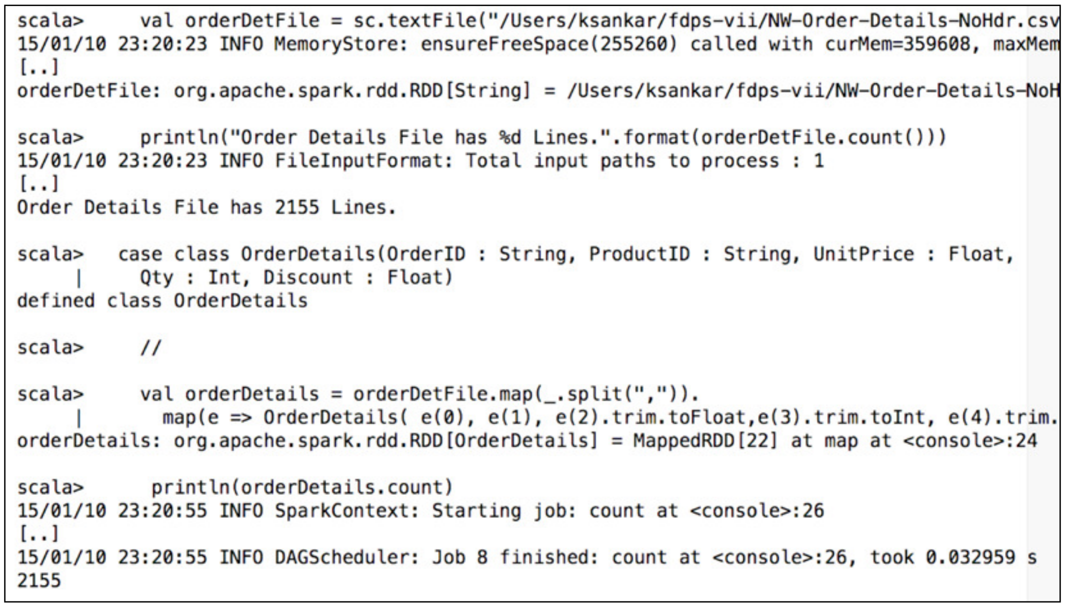


```
scala>      orders.registerTempTable("Orders")

scala>      var result = sqlContext.sql("SELECT * from Orders")
result: org.apache.spark.sql.SchemaRDD =
SchemaRDD[15] at RDD at SchemaRDD.scala:111
== Query Plan ==
== Physical Plan ==
PhysicalRDD [OrderID#11,CustomerID#12,EmployeeID#13,OrderDate#14,ShipCountry#15], MapParti

scala>      result.take(10).foreach(println)
15/01/10 23:11:43 INFO SparkContext: Starting job: runJob at basicOperators.scala:141
[..]
15/01/10 23:11:43 INFO DAGScheduler: Job 6 finished: runJob at basicOperators.scala:141, t
[10248,VINET,5,7/2/96,France]
[10249,TOMSP,6,7/3/96,Germany]
[10250,HANAR,4,7/6/96,Brazil]
[10251,VICTE,3,7/6/96,France]
[10252,SUPRD,4,7/7/96,Belgium]
[10253,HANAR,3,7/8/96,Brazil]
[10254,CHOPS,5,7/9/96,Switzerland]
[10255,RICSU,9,7/10/96,Switzerland]
[10256,WELLI,3,7/13/96,Brazil]
[10257,HILAA,4,7/14/96,Venezuela]
```

Now let's load the `Order Details` table. By now, we are an old hand at doing this. The following is the code for the loading process of the table:

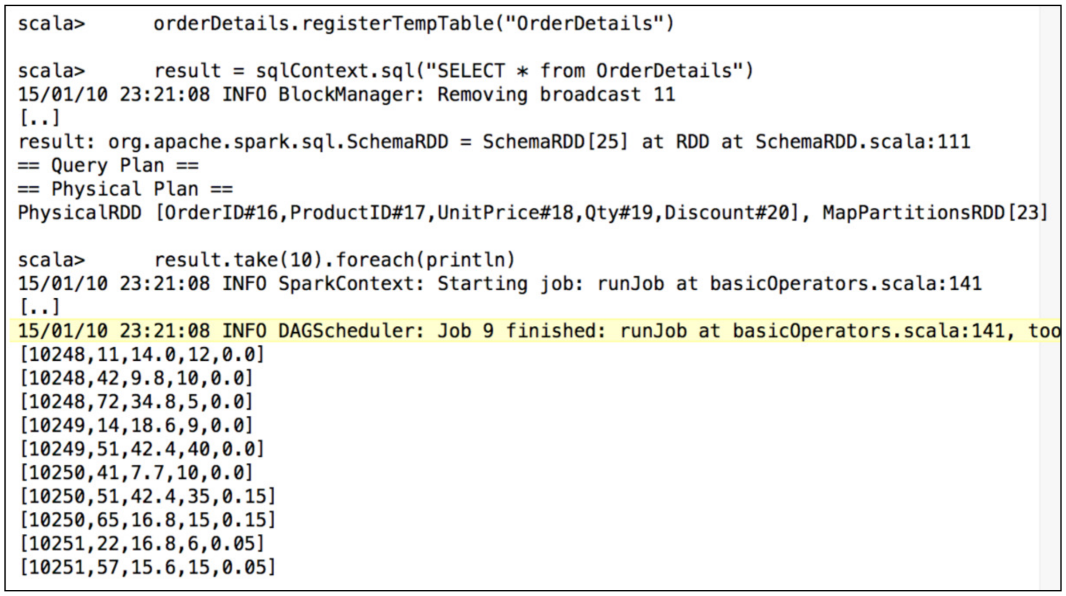
```
val orderDetFile = sc.textFile("/Users/ksankar/fdps-vii/NW-Order-
  Details-NoHdr.csv")
    println("Order Details File has %d Lines."
      .format(orderDetFile.count()))
val orderDetails = orderDetFile.map(_.split(",")).
      map(e => OrderDetails( e(0), e(1), e(2).
        trim.toFloat,e(3).trim.toInt, e(4).trim.toFloat ))
     println(orderDetails.count)
     orderDetails.registerTempTable("OrderDetails")
     result = sqlContext.sql("SELECT * from OrderDetails")
     result.take(10).foreach(println)
```

The output from the Spark shell is again as expected. It has 2,155 order details, as shown in the next screenshot:



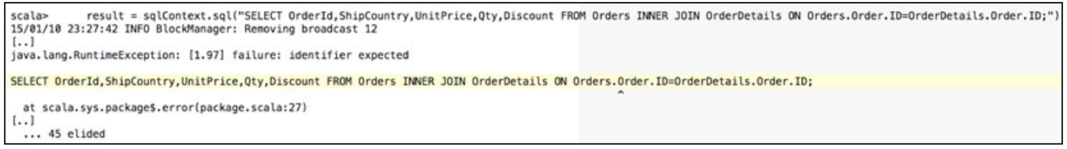
```
scala>     val orderDetFile = sc.textFile("/Users/ksankar/fdps-vii/NW-Order-Details-NoHdr.csv
15/01/10 23:20:23 INFO MemoryStore: ensureFreeSpace(255260) called with curMem=359608, maxMem
[..]
orderDetFile: org.apache.spark.rdd.RDD[String] = /Users/ksankar/fdps-vii/NW-Order-Details-NoH

scala>     println("Order Details File has %d Lines.".format(orderDetFile.count()))
15/01/10 23:20:23 INFO FileInputFormat: Total input paths to process : 1
[..]
Order Details File has 2155 Lines.

scala>   case class OrderDetails(OrderID : String, ProductID : String, UnitPrice : Float,
     |     Qty : Int, Discount : Float)
defined class OrderDetails

scala>     //

scala>     val orderDetails = orderDetFile.map(_.split(",")).
     |       map(e => OrderDetails( e(0), e(1), e(2).trim.toFloat,e(3).trim.toInt, e(4).trim.
orderDetails: org.apache.spark.rdd.RDD[OrderDetails] = MappedRDD[22] at map at <console>:24

scala>      println(orderDetails.count)
15/01/10 23:20:55 INFO SparkContext: Starting job: count at <console>:26
[..]
15/01/10 23:20:55 INFO DAGScheduler: Job 8 finished: count at <console>:26, took 0.032959 s
2155
```

Let's create the `Orderdetails` table and make sure it works as expected. The table is shown here:



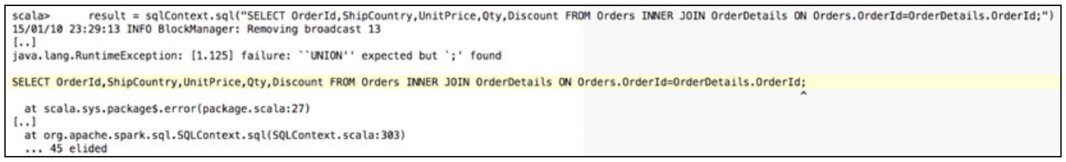
```
scala>      orderDetails.registerTempTable("OrderDetails")

scala>      result = sqlContext.sql("SELECT * from OrderDetails")
15/01/10 23:21:08 INFO BlockManager: Removing broadcast 11
[..]
result: org.apache.spark.sql.SchemaRDD = SchemaRDD[25] at RDD at SchemaRDD.scala:111
== Query Plan ==
== Physical Plan ==
PhysicalRDD [OrderID#16,ProductID#17,UnitPrice#18,Qty#19,Discount#20], MapPartitionsRDD[23]

scala>      result.take(10).foreach(println)
15/01/10 23:21:08 INFO SparkContext: Starting job: runJob at basicOperators.scala:141
[..]
15/01/10 23:21:08 INFO DAGScheduler: Job 9 finished: runJob at basicOperators.scala:141, too
[10248,11,14.0,12,0.0]
[10248,42,9.8,10,0.0]
[10248,72,34.8,5,0.0]
[10249,14,18.6,9,0.0]
[10249,51,42.4,40,0.0]
[10250,41,7.7,10,0.0]
[10250,51,42.4,35,0.15]
[10250,65,16.8,15,0.15]
[10251,22,16.8,6,0.05]
[10251,57,15.6,15,0.05]
```

Now comes the interesting part. Let's join the two tables and see how that query works. In this process, you might make some mistakes and learn a few things. Have a look at the following screenshot:



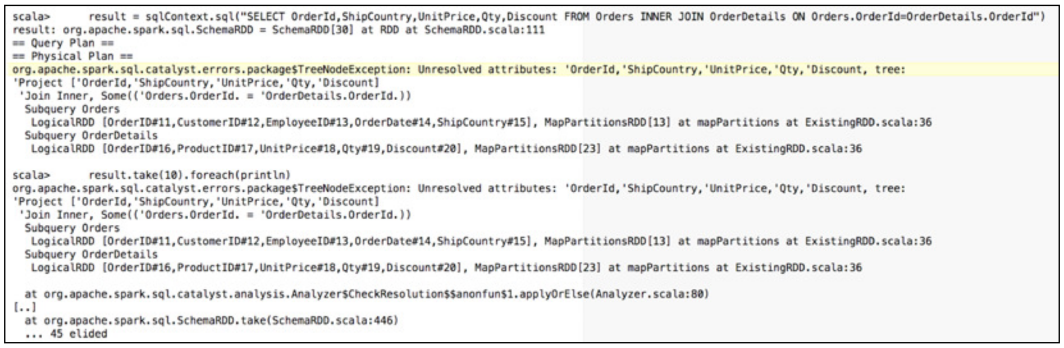
```
scala>      result = sqlContext.sql("SELECT OrderId,ShipCountry,UnitPrice,Qty,Discount FROM Orders INNER JOIN OrderDetails ON Orders.Order.ID=OrderDetails.Order.ID;")
15/01/10 23:27:42 INFO BlockManager: Removing broadcast 12
[..]
java.lang.RuntimeException: [1.97] failure: identifier expected

SELECT OrderId,ShipCountry,UnitPrice,Qty,Discount FROM Orders INNER JOIN OrderDetails ON Orders.Order.ID=OrderDetails.Order.ID;
                                                                                                ^

  at scala.sys.package$.error(package.scala:27)
[..]
  ... 45 elided
```

Here, the error was that `Order.ID` is a wrong name. So, we get the `identifier expected` error. Have a look at the following screenshot:



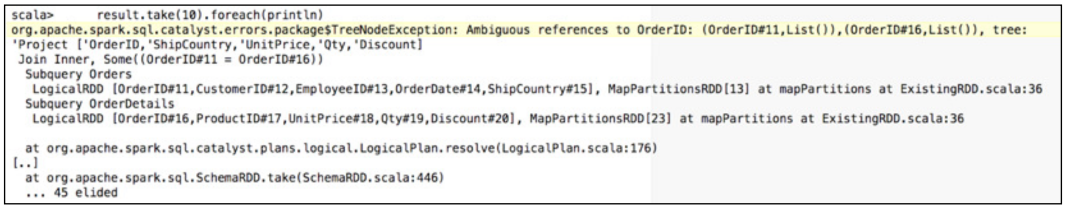
```
scala>      result = sqlContext.sql("SELECT OrderId,ShipCountry,UnitPrice,Qty,Discount FROM Orders INNER JOIN OrderDetails ON Orders.OrderId=OrderDetails.OrderId;")
15/01/10 23:29:13 INFO BlockManager: Removing broadcast 13
[..]
java.lang.RuntimeException: [1.125] failure: ``UNION'' expected but `;' found

SELECT OrderId,ShipCountry,UnitPrice,Qty,Discount FROM Orders INNER JOIN OrderDetails ON Orders.OrderId=OrderDetails.OrderId;
                                                                                                                            ^

  at scala.sys.package$.error(package.scala:27)
[..]
  at org.apache.spark.sql.SQLContext.sql(SQLContext.scala:303)
  ... 45 elided
```

This is interesting. It doesn't like the `;` at the end!. Now, have a look at the following screenshot:



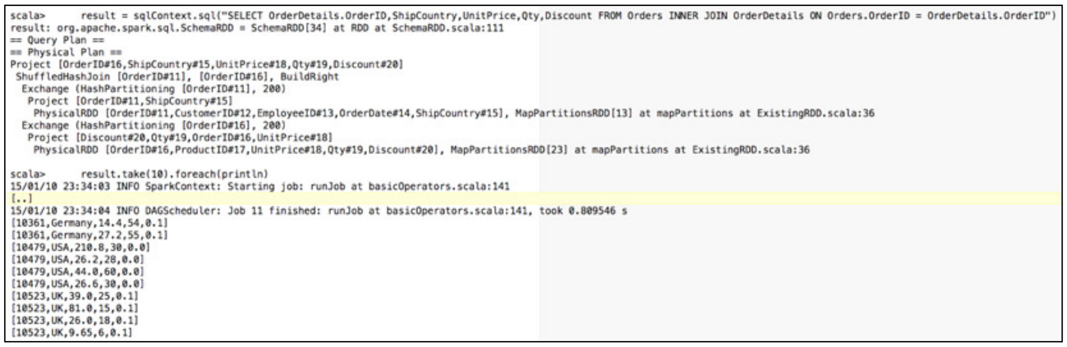
```
scala>      result = sqlContext.sql("SELECT OrderId,ShipCountry,UnitPrice,Qty,Discount FROM Orders INNER JOIN OrderDetails ON Orders.OrderId=OrderDetails.OrderId")
result: org.apache.spark.sql.SchemaRDD = SchemaRDD[30] at RDD at SchemaRDD.scala:111
== Query Plan ==
== Physical Plan ==
org.apache.spark.sql.catalyst.errors.package$TreeNodeException: Unresolved attributes: 'OrderId,'ShipCountry,'UnitPrice,'Qty,'Discount, tree:
'Project ['OrderId,'ShipCountry,'UnitPrice,'Qty,'Discount]
 'Join Inner, Some(('Orders.OrderId. = 'OrderDetails.OrderId.))
  Subquery Orders
   LogicalRDD [OrderID#11,CustomerID#12,EmployeeID#13,OrderDate#14,ShipCountry#15], MapPartitionsRDD[13] at mapPartitions at ExistingRDD.scala:36
  Subquery OrderDetails
   LogicalRDD [OrderID#16,ProductID#17,UnitPrice#18,Qty#19,Discount#20], MapPartitionsRDD[23] at mapPartitions at ExistingRDD.scala:36

scala>      result.take(10).foreach(println)
org.apache.spark.sql.catalyst.errors.package$TreeNodeException: Unresolved attributes: 'OrderId,'ShipCountry,'UnitPrice,'Qty,'Discount, tree:
'Project ['OrderId,'ShipCountry,'UnitPrice,'Qty,'Discount]
 'Join Inner, Some(('Orders.OrderId. = 'OrderDetails.OrderId.))
  Subquery Orders
   LogicalRDD [OrderID#11,CustomerID#12,EmployeeID#13,OrderDate#14,ShipCountry#15], MapPartitionsRDD[13] at mapPartitions at ExistingRDD.scala:36
  Subquery OrderDetails
   LogicalRDD [OrderID#16,ProductID#17,UnitPrice#18,Qty#19,Discount#20], MapPartitionsRDD[23] at mapPartitions at ExistingRDD.scala:36

  at org.apache.spark.sql.catalyst.analysis.Analyzer$CheckResolution$$anonfun$1.applyOrElse(Analyzer.scala:80)
[..]
  at org.apache.spark.sql.SchemaRDD.take(SchemaRDD.scala:446)
  ... 45 elided
```

This one took me a little time to figure out. The culprit was `OrderId`, which is really `OrderID`!. Now, consider the following screenshot:



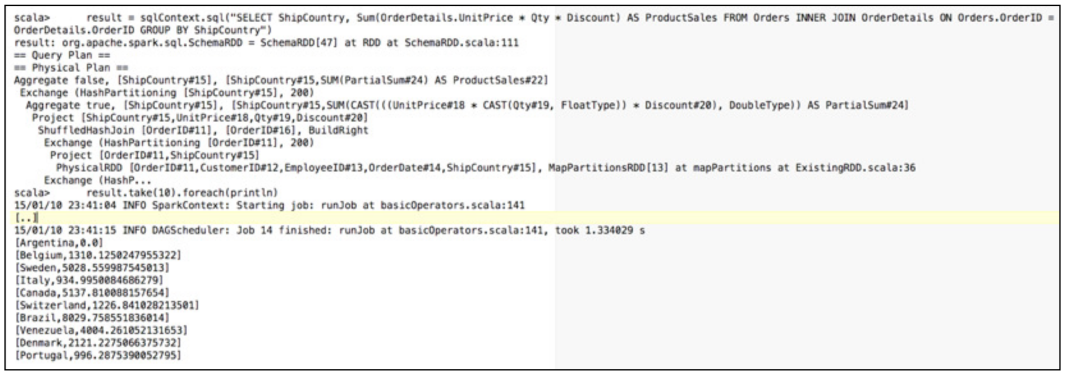
```
scala>      result.take(10).foreach(println)
org.apache.spark.sql.catalyst.errors.package$TreeNodeException: Ambiguous references to OrderID: (OrderID#11,List()),(OrderID#16,List()), tree:
'Project ['OrderID,'ShipCountry,'UnitPrice,'Qty,'Discount]
 Join Inner, Some((OrderID#11 = OrderID#16))
  Subquery Orders
   LogicalRDD [OrderID#11,CustomerID#12,EmployeeID#13,OrderDate#14,ShipCountry#15], MapPartitionsRDD[13] at mapPartitions at ExistingRDD.scala:36
  Subquery OrderDetails
   LogicalRDD [OrderID#16,ProductID#17,UnitPrice#18,Qty#19,Discount#20], MapPartitionsRDD[23] at mapPartitions at ExistingRDD.scala:36

  at org.apache.spark.sql.catalyst.plans.logical.LogicalPlan.resolve(LogicalPlan.scala:176)
[..]
  at org.apache.spark.sql.SchemaRDD.take(SchemaRDD.scala:446)
  ... 45 elided
```

Now it understands all the attributes. Of course, there are two `OrderID` values, one from the `Orders` table and another from the `OrderDetails` table. Have a look at the next screenshot:



```
scala>      result = sqlContext.sql("SELECT OrderDetails.OrderID,ShipCountry,UnitPrice,Qty,Discount FROM Orders INNER JOIN OrderDetails ON Orders.OrderID = OrderDetails.OrderID")
result: org.apache.spark.sql.SchemaRDD = SchemaRDD[34] at RDD at SchemaRDD.scala:111
== Query Plan ==
== Physical Plan ==
Project [OrderID#16,ShipCountry#15,UnitPrice#18,Qty#19,Discount#20]
 ShuffledHashJoin [OrderID#11], [OrderID#16], BuildRight
  Exchange (HashPartitioning [OrderID#11], 200)
   Project [OrderID#11,ShipCountry#15]
    PhysicalRDD [OrderID#11,CustomerID#12,EmployeeID#13,OrderDate#14,ShipCountry#15], MapPartitionsRDD[13] at mapPartitions at ExistingRDD.scala:36
  Exchange (HashPartitioning [OrderID#16], 200)
   Project [Discount#20,Qty#19,OrderID#16,UnitPrice#18]
    PhysicalRDD [OrderID#16,ProductID#17,UnitPrice#18,Qty#19,Discount#20], MapPartitionsRDD[23] at mapPartitions at ExistingRDD.scala:36

scala>      result.take(10).foreach(println)
15/01/10 23:34:03 INFO SparkContext: Starting job: runJob at basicOperators.scala:141
[..]
15/01/10 23:34:04 INFO DAGScheduler: Job 11 finished: runJob at basicOperators.scala:141, took 0.809546 s
[10361,Germany,14.4,54,0.1]
[10361,Germany,27.2,55,0.1]
[10479,USA,210.8,30,0.0]
[10479,USA,26.2,28,0.0]
[10479,USA,44.0,60,0.0]
[10479,USA,26.6,30,0.0]
[10523,UK,39.0,25,0.1]
[10523,UK,81.0,15,0.1]
[10523,UK,26.0,18,0.1]
[10523,UK,9.65,6,0.1]
```

Finally, after correcting the errors, it works fine! Good stuff! Now, have a look at the following screenshot:

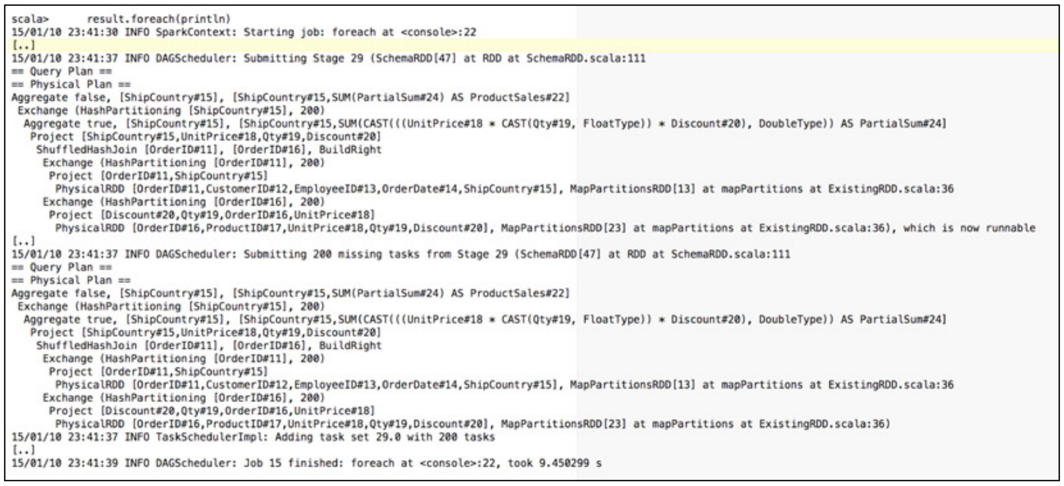


```
scala>     result = sqlContext.sql("SELECT ShipCountry, Sum(OrderDetails.UnitPrice * Qty * Discount) AS ProductSales FROM Orders INNER JOIN OrderDetails ON Orders.OrderID =
OrderDetails.OrderID GROUP BY ShipCountry")
result: org.apache.spark.sql.SchemaRDD = SchemaRDD[47] at RDD at SchemaRDD.scala:111
== Query Plan ==
== Physical Plan ==
Aggregate false, [ShipCountry#15], [ShipCountry#15,SUM(PartialSum#24) AS ProductSales#22]
 Exchange (HashPartitioning [ShipCountry#15], 200)
  Aggregate true, [ShipCountry#15], [ShipCountry#15,SUM(CAST(((UnitPrice#18 * CAST(Qty#19, FloatType)) * Discount#20), DoubleType)) AS PartialSum#24]
   Project [ShipCountry#15,UnitPrice#18,Qty#19,Discount#20]
    ShuffledHashJoin [OrderID#11], [OrderID#16], BuildRight
     Exchange (HashPartitioning [OrderID#11], 200)
      Project [OrderID#11,ShipCountry#15]
       PhysicalRDD [OrderID#11,CustomerID#12,EmployeeID#13,OrderDate#14,ShipCountry#15], MapPartitionsRDD[13] at mapPartitions at ExistingRDD.scala:36
     Exchange (HashP...
scala>     result.take(10).foreach(println)
15/01/10 23:41:04 INFO SparkContext: Starting job: runJob at basicOperators.scala:141
[..]
15/01/10 23:41:15 INFO DAGScheduler: Job 14 finished: runJob at basicOperators.scala:141, took 1.334029 s
[Argentina,0.0]
[Belgium,1310.1250247955322]
[Sweden,5028.559987545013]
[Italy,934.9950084686279]
[Canada,5137.810088157654]
[Switzerland,1226.841028213501]
[Brazil,8029.758551836014]
[Venezuela,4004.261052131653]
[Denmark,2121.2275066375732]
[Portugal,996.2875390052795]
```

Interestingly, this worked on the first try, the credit for which goes to the Spark developers. In my machine, Spark progressively spawned many tasks with lots of shuffle and broadcast stages. You will see so many pages of logs entries (approximately 2,500 lines!); we suggest you just quickly browse through them to get a feel for the workflow graph.

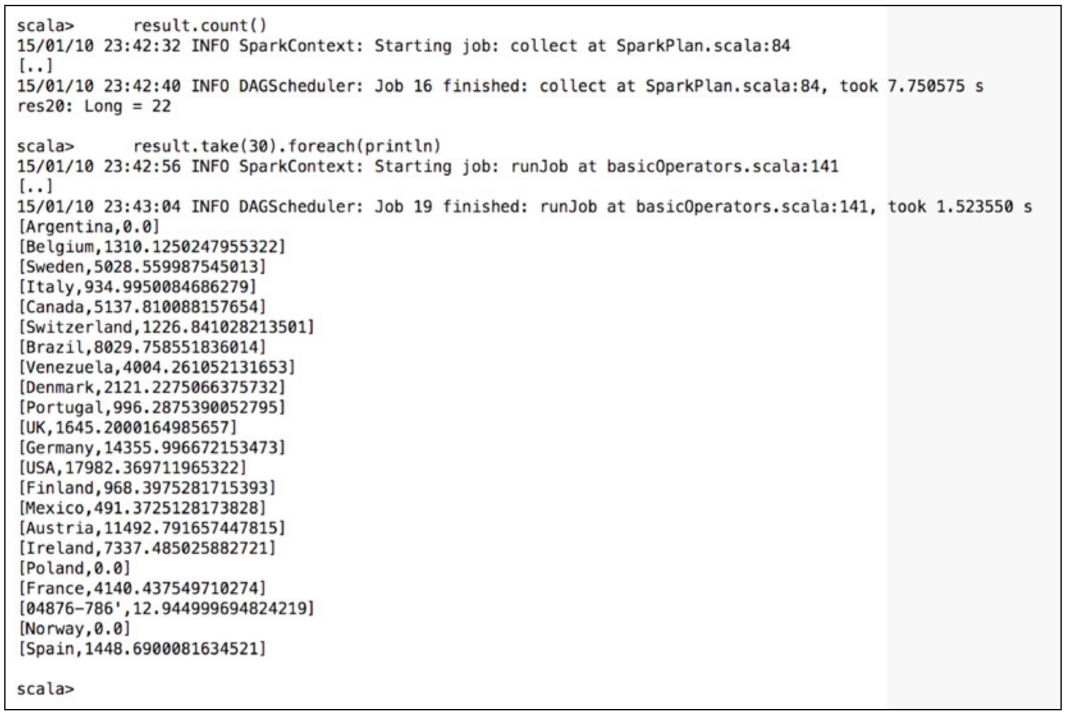
Before we end this chapter, let us try printing all the results. The call skips `take(10)`.

The `scala> result.foreach(println)` command works fine, but the results were mixed with the log entries. Take a quick look at the query plan it has printed out. It gives us an insight on the different operations it performs on the RDD, as shown in the next screenshot:



```
scala>     result.foreach(println)
15/01/10 23:41:30 INFO SparkContext: Starting job: foreach at <console>:22
[..]
15/01/10 23:41:37 INFO DAGScheduler: Submitting Stage 29 (SchemaRDD[47] at RDD at SchemaRDD.scala:111
== Query Plan ==
== Physical Plan ==
Aggregate false, [ShipCountry#15], [ShipCountry#15,SUM(PartialSum#24) AS ProductSales#22]
 Exchange (HashPartitioning [ShipCountry#15], 200)
  Aggregate true, [ShipCountry#15], [ShipCountry#15,SUM(CAST(((UnitPrice#18 * CAST(Qty#19, FloatType)) * Discount#20), DoubleType)) AS PartialSum#24]
   Project [ShipCountry#15,UnitPrice#18,Qty#19,Discount#20]
    ShuffledHashJoin [OrderID#11], [OrderID#16], BuildRight
     Exchange (HashPartitioning [OrderID#11], 200)
      Project [OrderID#11,ShipCountry#15]
       PhysicalRDD [OrderID#11,CustomerID#12,EmployeeID#13,OrderDate#14,ShipCountry#15], MapPartitionsRDD[13] at mapPartitions at ExistingRDD.scala:36
     Exchange (HashPartitioning [OrderID#16], 200)
      Project [Discount#20,Qty#19,OrderID#16,UnitPrice#18]
       PhysicalRDD [OrderID#16,ProductID#17,UnitPrice#18,Qty#19,Discount#20], MapPartitionsRDD[23] at mapPartitions at ExistingRDD.scala:36), which is now runnable
[..]
15/01/10 23:41:37 INFO DAGScheduler: Submitting 200 missing tasks from Stage 29 (SchemaRDD[47] at RDD at SchemaRDD.scala:111
== Query Plan ==
== Physical Plan ==
Aggregate false, [ShipCountry#15], [ShipCountry#15,SUM(PartialSum#24) AS ProductSales#22]
 Exchange (HashPartitioning [ShipCountry#15], 200)
  Aggregate true, [ShipCountry#15], [ShipCountry#15,SUM(CAST(((UnitPrice#18 * CAST(Qty#19, FloatType)) * Discount#20), DoubleType)) AS PartialSum#24]
   Project [ShipCountry#15,UnitPrice#18,Qty#19,Discount#20]
    ShuffledHashJoin [OrderID#11], [OrderID#16], BuildRight
     Exchange (HashPartitioning [OrderID#11], 200)
      Project [OrderID#11,ShipCountry#15]
       PhysicalRDD [OrderID#11,CustomerID#12,EmployeeID#13,OrderDate#14,ShipCountry#15], MapPartitionsRDD[13] at mapPartitions at ExistingRDD.scala:36
     Exchange (HashPartitioning [OrderID#16], 200)
      Project [Discount#20,Qty#19,OrderID#16,UnitPrice#18]
       PhysicalRDD [OrderID#16,ProductID#17,UnitPrice#18,Qty#19,Discount#20], MapPartitionsRDD[23] at mapPartitions at ExistingRDD.scala:36)
15/01/10 23:41:37 INFO TaskSchedulerImpl: Adding task set 29.0 with 200 tasks
[..]
15/01/10 23:41:39 INFO DAGScheduler: Job 15 finished: foreach at <console>:22, took 9.450299 s
```

I did a count and then printed out all the records, as shown in the next screenshot. It worked out well. We could also format the printout with currency as well. I leave that as an exercise to be done by you!



```
scala>      result.count()
15/01/10 23:42:32 INFO SparkContext: Starting job: collect at SparkPlan.scala:84
[..]
15/01/10 23:42:40 INFO DAGScheduler: Job 16 finished: collect at SparkPlan.scala:84, took 7.750575 s
res20: Long = 22

scala>      result.take(30).foreach(println)
15/01/10 23:42:56 INFO SparkContext: Starting job: runJob at basicOperators.scala:141
[..]
15/01/10 23:43:04 INFO DAGScheduler: Job 19 finished: runJob at basicOperators.scala:141, took 1.523550 s
[Argentina,0.0]
[Belgium,1310.1250247955322]
[Sweden,5028.559987545013]
[Italy,934.9950084686279]
[Canada,5137.810088157654]
[Switzerland,1226.841028213501]
[Brazil,8029.758551836014]
[Venezuela,4004.261052131653]
[Denmark,2121.2275066375732]
[Portugal,996.2875390052795]
[UK,1645.2000164985657]
[Germany,14355.996672153473]
[USA,17982.369711965322]
[Finland,968.3975281715393]
[Mexico,491.3725128173828]
[Austria,11492.791657447815]
[Ireland,7337.485025882721]
[Poland,0.0]
[France,4140.437549710274]
[04876-786',12.944999694824219]
[Norway,0.0]
[Spain,1448.6900081634521]

scala>
```

Aftermath

As seen in the preceding screenshot, this was a good exercise. We are thoroughly impressed! We just created the last query and it ran fine! The Spark developers have done a good job. Good work, guys.

The dataset also includes the `product` table, which I leave to you as an exercise. For example, you can work on a query that gives the sales by product or one that shows which products are selling more. The dataset also has date fields such as `order dates`, which you can use to query sales by quarter or reports like *Product sales for 1997*. The dates are now read in as strings. They need to be converted to the `TIMESTAMP` data type.

Some more information can be found at the following sites:

- `https://northwinddatabase.codeplex.com/releases/view/71634`
- `https://databricks.com/blog/2015/01/09/spark-sql-data-sources-api-unified-data-access-for-the-spark-platform.html`
- `https://spark.apache.org/docs/latest/sql-programming-guide.html`

Summary

This was an important chapter that discussed the integration aspects of Spark. We have covered the main parts, namely, SchemaRDD and programmatic access. But there are more capabilities such as the JDBC/ODBC server for direct SQL queries as well as the Spark SQL CLI. On the integration side, you will see more integration capabilities in *Chapter 8*, *Spark with Big Data*. Spark SQL will be getting more features in future versions and I think this will be one of the areas that will grow at a much faster pace; interesting features such as partitioning, persistent tables, and optional user specified schema are slated for Spark 1.3.

8
Spark with Big Data

As we mentioned in *Chapter 7, Spark SQL*, the big data compute stack doesn't work in isolation. Integration points across multiple stacks and technologies are essential. In this chapter, we will look at how Spark works with some of the big data technologies that are part of the Hadoop ecosystem. We will cover the following topics in this chapter:

- **Parquet**: This is an efficient storage format
- **HBase**: This is the database in the Hadoop Ecosystem

Parquet – an efficient and interoperable big data format

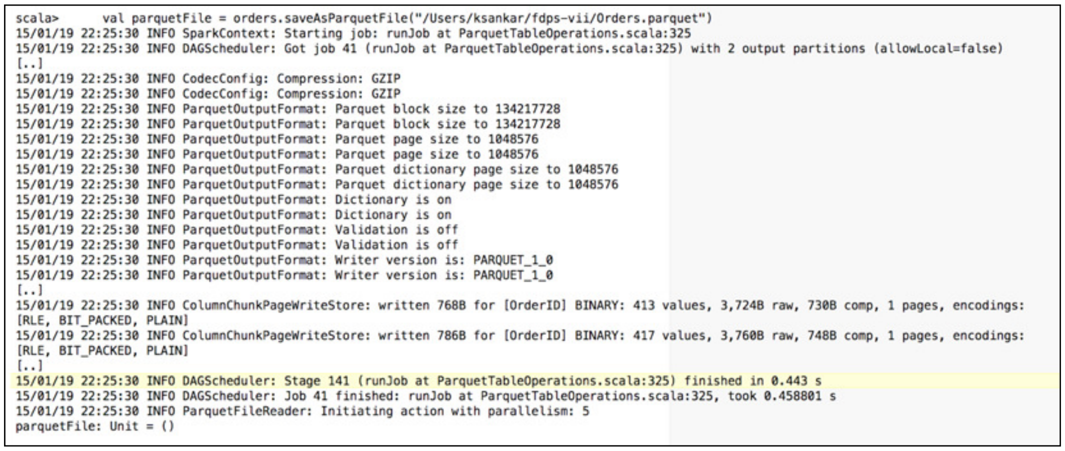
Parquet is essentially an interoperable storage format; its main goals are space efficiency and query efficiency. Parquet's origin is based on Google's Dremel and was developed by Twitter and Cloudera. Parquet is now an Apache incubator project. The nested storage format from Google Dremel is implemented in Parquet. Parquet stores data in a columnar format and has an evolvable schema. This enables you to optimize queries (it can restrict columns that you need to access, and so you need not bring all columns into memory and discard the ones not needed), and it allows storage optimization (by decoding at the column level, which gives a much higher compression ratio). In addition to the ability to restrict column fetches during queries, Parquet 2.0 would implement push-down predicates. While writing this book, the Parquet version was 1.6.

Saving files to the Parquet format

In *Chapter 7*, *Spark SQL*, we loaded the `Orders` tables from the `.csv` format. Let's save the data in the Parquet format so that we can query the data from Impala. Usually one would take a `.csv` file, do transformations, and then store it in the Parquet format (for example, the Sales By Country RDD that we had created). This is shown here:

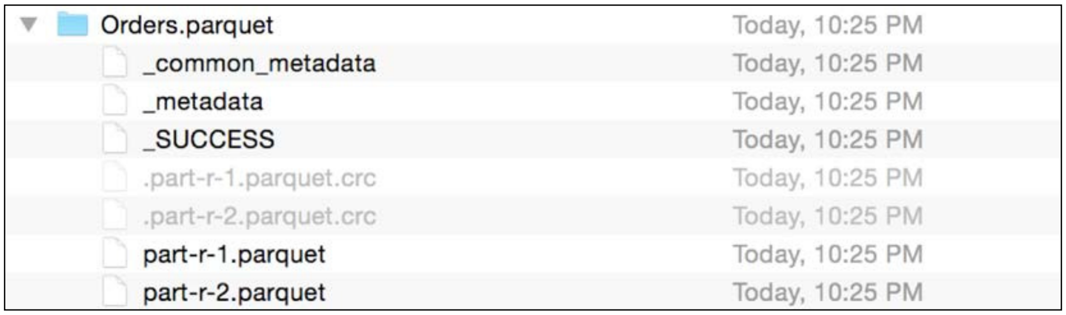
```
    //
    // Parquet Operations
    //
valparquetFileOrders     val parquetFileOrders = orders.
saveAsParquetFile("/Users/ksankar/fdps-vii/Orders.parquet")
valparquetFileOrderDet     val parquetFileOrderDet = orderDetails.
saveAsParquetFile("/Users/ksankar/fdps-vii/OrderDetails.parquet")
```

The output is shown in the following screenshot:



```
scala>     val parquetFile = orders.saveAsParquetFile("/Users/ksankar/fdps-vii/Orders.parquet")
15/01/19 22:25:30 INFO SparkContext: Starting job: runJob at ParquetTableOperations.scala:325
15/01/19 22:25:30 INFO DAGScheduler: Got job 41 (runJob at ParquetTableOperations.scala:325) with 2 output partitions (allowLocal=false)
[..]
15/01/19 22:25:30 INFO CodecConfig: Compression: GZIP
15/01/19 22:25:30 INFO CodecConfig: Compression: GZIP
15/01/19 22:25:30 INFO ParquetOutputFormat: Parquet block size to 134217728
15/01/19 22:25:30 INFO ParquetOutputFormat: Parquet block size to 134217728
15/01/19 22:25:30 INFO ParquetOutputFormat: Parquet page size to 1048576
15/01/19 22:25:30 INFO ParquetOutputFormat: Parquet page size to 1048576
15/01/19 22:25:30 INFO ParquetOutputFormat: Parquet dictionary page size to 1048576
15/01/19 22:25:30 INFO ParquetOutputFormat: Parquet dictionary page size to 1048576
15/01/19 22:25:30 INFO ParquetOutputFormat: Dictionary is on
15/01/19 22:25:30 INFO ParquetOutputFormat: Dictionary is on
15/01/19 22:25:30 INFO ParquetOutputFormat: Validation is off
15/01/19 22:25:30 INFO ParquetOutputFormat: Validation is off
15/01/19 22:25:30 INFO ParquetOutputFormat: Writer version is: PARQUET_1_0
15/01/19 22:25:30 INFO ParquetOutputFormat: Writer version is: PARQUET_1_0
[..]
15/01/19 22:25:30 INFO ColumnChunkPageWriteStore: written 768B for [OrderID] BINARY: 413 values, 3,724B raw, 730B comp, 1 pages, encodings:
[RLE, BIT_PACKED, PLAIN]
15/01/19 22:25:30 INFO ColumnChunkPageWriteStore: written 786B for [OrderID] BINARY: 417 values, 3,760B raw, 748B comp, 1 pages, encodings:
[RLE, BIT_PACKED, PLAIN]
[..]
15/01/19 22:25:30 INFO DAGScheduler: Stage 141 (runJob at ParquetTableOperations.scala:325) finished in 0.443 s
15/01/19 22:25:30 INFO DAGScheduler: Job 41 finished: runJob at ParquetTableOperations.scala:325, took 0.458801 s
15/01/19 22:25:30 INFO ParquetFileReader: Initiating action with parallelism: 5
parquetFile: Unit = ()
```

Even though in this example we store the Parquet file in the local filesystem, in the actual production system you would use HDFS to store the files. We can inspect the log entries and see that it has started a job with the `ParquetTableOperations` class. The scheme used to save this was **Run Length Encoding** (**RLE**). As you can see, we need only a couple of lines of code and Spark does all the hard work under the covers. It creates a directory, data, and metadata files underneath the main directory. It has created two files corresponding to the two jobs for two partitions, as shown here:



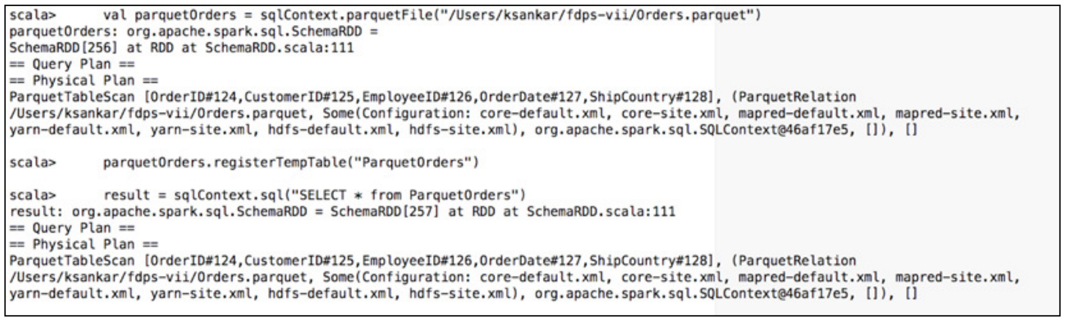
```
Orders.parquet              Today, 10:25 PM
  _common_metadata          Today, 10:25 PM
  _metadata                 Today, 10:25 PM
  _SUCCESS                  Today, 10:25 PM
  .part-r-1.parquet.crc     Today, 10:25 PM
  .part-r-2.parquet.crc     Today, 10:25 PM
  part-r-1.parquet          Today, 10:25 PM
  part-r-2.parquet          Today, 10:25 PM
```

Loading Parquet files

Let's now load the `Orders` Parquet files and see whether the data got saved correctly. The code, again, is deceptively simple, as shown here:

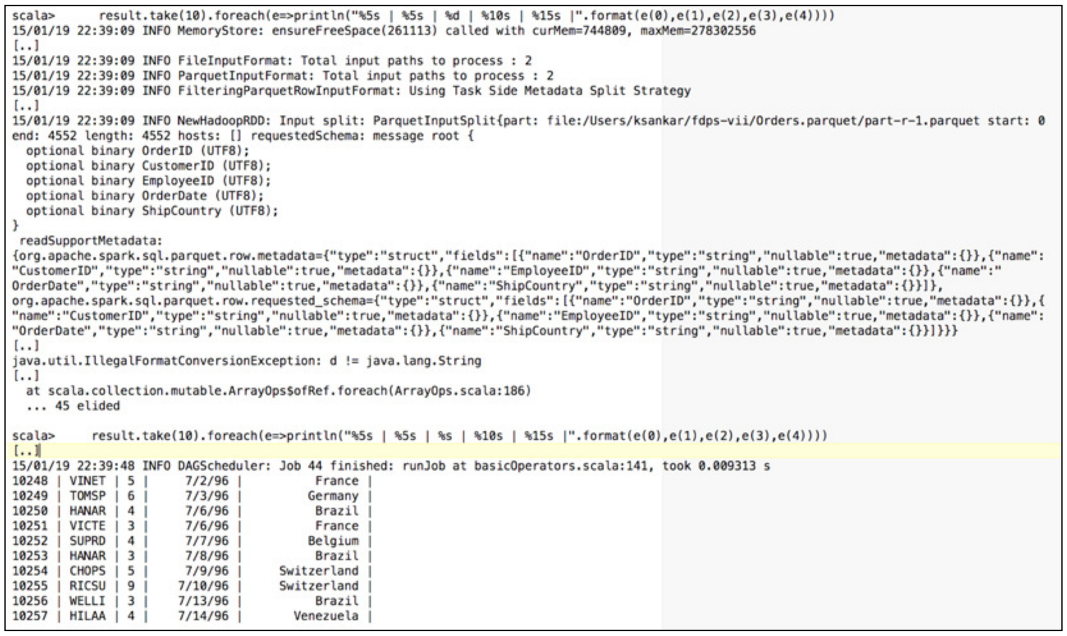
```
     //
     // Let us read back the file
     //
valsqlContext= new org.apache.spark.sql.SQLContext(sc)
val parquetOrders= sqlContext.parquetFile("/Users/ksankar/fdps-vii/
Orders.parquet")
     parquetOrders.registerTempTable("ParquetOrders")
     val result = sqlContext.sql("SELECT * from ParquetOrders")
     result.take(10).foreach(e=>println("%5s | %5s | %s | %10s | %15s
|".format(e(0),e(1),e(2),e(3),e(4))))
```

The output is as follows:



```
scala>      val parquetOrders = sqlContext.parquetFile("/Users/ksankar/fdps-vii/Orders.parquet")
parquetOrders: org.apache.spark.sql.SchemaRDD =
SchemaRDD[256] at RDD at SchemaRDD.scala:111
== Query Plan ==
== Physical Plan ==
ParquetTableScan [OrderID#124,CustomerID#125,EmployeeID#126,OrderDate#127,ShipCountry#128], (ParquetRelation
/Users/ksankar/fdps-vii/Orders.parquet, Some(Configuration: core-default.xml, core-site.xml, mapred-default.xml, mapred-site.xml,
yarn-default.xml, yarn-site.xml, hdfs-default.xml, hdfs-site.xml), org.apache.spark.sql.SQLContext@46af17e5, []), []

scala>      parquetOrders.registerTempTable("ParquetOrders")

scala>      result = sqlContext.sql("SELECT * from ParquetOrders")
result: org.apache.spark.sql.SchemaRDD = SchemaRDD[257] at RDD at SchemaRDD.scala:111
== Query Plan ==
== Physical Plan ==
ParquetTableScan [OrderID#124,CustomerID#125,EmployeeID#126,OrderDate#127,ShipCountry#128], (ParquetRelation
/Users/ksankar/fdps-vii/Orders.parquet, Some(Configuration: core-default.xml, core-site.xml, mapred-default.xml, mapred-site.xml,
yarn-default.xml, yarn-site.xml, hdfs-default.xml, hdfs-site.xml), org.apache.spark.sql.SQLContext@46af17e5, []), []
```

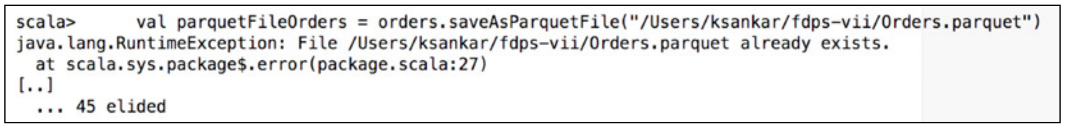
As you can see, the first few lines create all the scaffolding and needed definitions. The lazy evaluation does not do anything unless we ask for some action, such as `take(10)`, as shown in the following screenshot:



```
scala>     result.take(10).foreach(e=>println("%5s | %5s | %d | %10s | %15s |".format(e(0),e(1),e(2),e(3),e(4))))
15/01/19 22:39:09 INFO MemoryStore: ensureFreeSpace(261113) called with curMem=744809, maxMem=278302556
[..]
15/01/19 22:39:09 INFO FileInputFormat: Total input paths to process : 2
15/01/19 22:39:09 INFO ParquetInputFormat: Total input paths to process : 2
15/01/19 22:39:09 INFO FilteringParquetRowInputFormat: Using Task Side Metadata Split Strategy
[..]
15/01/19 22:39:09 INFO NewHadoopRDD: Input split: ParquetInputSplit{part: file:/Users/ksankar/fdps-vii/Orders.parquet/part-r-1.parquet start: 0
end: 4552 length: 4552 hosts: [] requestedSchema: message root {
  optional binary OrderID (UTF8);
  optional binary CustomerID (UTF8);
  optional binary EmployeeID (UTF8);
  optional binary OrderDate (UTF8);
  optional binary ShipCountry (UTF8);
}
 readSupportMetadata:
{org.apache.spark.sql.parquet.row.metadata={"type":"struct","fields":[{"name":"OrderID","type":"string","nullable":true,"metadata":{}},{"name":
"CustomerID","type":"string","nullable":true,"metadata":{}},{"name":"EmployeeID","type":"string","nullable":true,"metadata":{}},{"name":"
OrderDate","type":"string","nullable":true,"metadata":{}},{"name":"ShipCountry","type":"string","nullable":true,"metadata":{}}]},
org.apache.spark.sql.parquet.row.requested_schema={"type":"struct","fields":[{"name":"OrderID","type":"string","nullable":true,"metadata":{}},{
"name":"CustomerID","type":"string","nullable":true,"metadata":{}},{"name":"EmployeeID","type":"string","nullable":true,"metadata":{}},{"name":
"OrderDate","type":"string","nullable":true,"metadata":{}},{"name":"ShipCountry","type":"string","nullable":true,"metadata":{}}]}}}
[..]
java.util.IllegalFormatConversionException: d != java.lang.String
[..]
  at scala.collection.mutable.ArrayOps$ofRef.foreach(ArrayOps.scala:186)
  ... 45 elided

scala>    result.take(10).foreach(e=>println("%5s | %5s | %s | %10s | %15s |".format(e(0),e(1),e(2),e(3),e(4))))
[..]
15/01/19 22:39:48 INFO DAGScheduler: Job 44 finished: runJob at basicOperators.scala:141, took 0.009313 s
10248 | VINET | 5 |     7/2/96 |          France |
10249 | TOMSP | 6 |     7/3/96 |         Germany |
10250 | HANAR | 4 |     7/6/96 |          Brazil |
10251 | VICTE | 3 |     7/6/96 |          France |
10252 | SUPRD | 4 |     7/7/96 |         Belgium |
10253 | HANAR | 3 |     7/8/96 |          Brazil |
10254 | CHOPS | 5 |     7/9/96 |     Switzerland |
10255 | RICSU | 9 |    7/10/96 |     Switzerland |
10256 | WELLI | 3 |    7/13/96 |          Brazil |
10257 | HILAA | 4 |    7/14/96 |       Venezuela |
```

It does all the work. You can see that Spark figured out that there are two files to process along with the field names and their types. It actually fails with an error, because `EmployeeID` was defined as string and I tried to print it with the `'%d'` mask. Now that's interesting, Spark keeps the data type in the Parquet metadata and can read it back. Once I used the `%s` mask, everything worked out fine.

Note that you cannot overwrite a Parquet file, as shown:



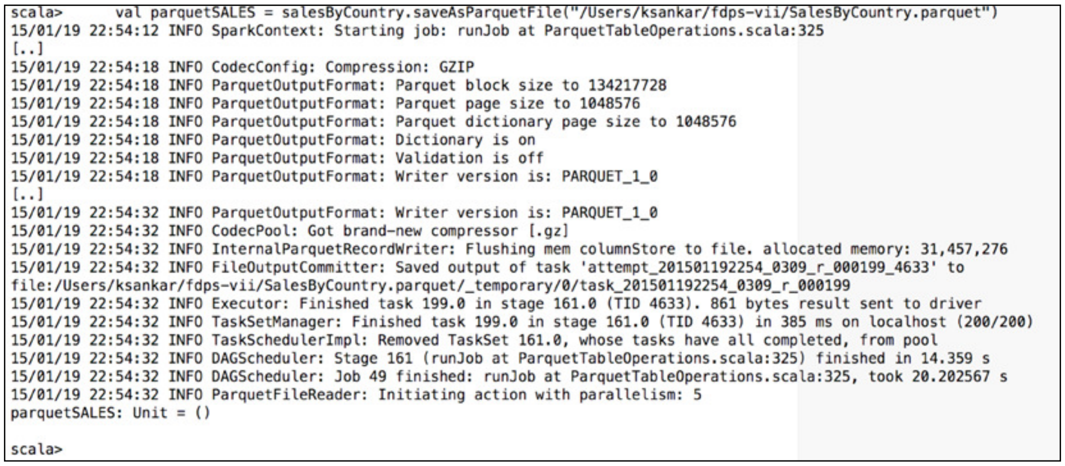
```
scala>     val parquetFileOrders = orders.saveAsParquetFile("/Users/ksankar/fdps-vii/Orders.parquet")
java.lang.RuntimeException: File /Users/ksankar/fdps-vii/Orders.parquet already exists.
  at scala.sys.package$.error(package.scala:27)
[..]
  ... 45 elided
```

Saving processed RDD in the Parquet format

Now let's save our `SalesByCountry` report in the Parquet format. We create a SQL table `ieaSchemaRDD` and then save that as a Parquet file:

```
        //
        // Save our Sales By Country Report as parquet
        //
   valsalesByCountry = sqlContext.sql("SELECT ShipCountry,
   Sum(OrderDetails.UnitPrice * Qty * Discount) AS ProductSales FROM
   Orders INNER JOIN OrderDetails ON Orders.OrderID = OrderDetails.
   OrderID GROUP BY ShipCountry")
        salesByCountry.registerTempTable("SalesByCountry")
        result = sqlContext.sql("SELECT * from SalesByCountry")
        result.take(30).foreach(e=>println("%15s | %9.2f
          |".format(e(0),e(1))))
   valparquetSALES = salesByCountry.saveAsParquetFile("/Users/ksankar/
   fdps-vii/SalesByCountry.parquet")
```

By now we know the drill, and as expected, the files are created, as shown next:



```
scala>      val parquetSALES = salesByCountry.saveAsParquetFile("/Users/ksankar/fdps-vii/SalesByCountry.parquet")
15/01/19 22:54:12 INFO SparkContext: Starting job: runJob at ParquetTableOperations.scala:325
[..]
15/01/19 22:54:18 INFO CodecConfig: Compression: GZIP
15/01/19 22:54:18 INFO ParquetOutputFormat: Parquet block size to 134217728
15/01/19 22:54:18 INFO ParquetOutputFormat: Parquet page size to 1048576
15/01/19 22:54:18 INFO ParquetOutputFormat: Parquet dictionary page size to 1048576
15/01/19 22:54:18 INFO ParquetOutputFormat: Dictionary is on
15/01/19 22:54:18 INFO ParquetOutputFormat: Validation is off
15/01/19 22:54:18 INFO ParquetOutputFormat: Writer version is: PARQUET_1_0
[..]
15/01/19 22:54:32 INFO ParquetOutputFormat: Writer version is: PARQUET_1_0
15/01/19 22:54:32 INFO CodecPool: Got brand-new compressor [.gz]
15/01/19 22:54:32 INFO InternalParquetRecordWriter: Flushing mem columnStore to file. allocated memory: 31,457,276
15/01/19 22:54:32 INFO FileOutputCommitter: Saved output of task 'attempt_201501192254_0309_r_000199_4633' to
file:/Users/ksankar/fdps-vii/SalesByCountry.parquet/_temporary/0/task_201501192254_0309_r_000199
15/01/19 22:54:32 INFO Executor: Finished task 199.0 in stage 161.0 (TID 4633). 861 bytes result sent to driver
15/01/19 22:54:32 INFO TaskSetManager: Finished task 199.0 in stage 161.0 (TID 4633) in 385 ms on localhost (200/200)
15/01/19 22:54:32 INFO TaskSchedulerImpl: Removed TaskSet 161.0, whose tasks have all completed, from pool
15/01/19 22:54:32 INFO DAGScheduler: Stage 161 (runJob at ParquetTableOperations.scala:325) finished in 14.359 s
15/01/19 22:54:32 INFO DAGScheduler: Job 49 finished: runJob at ParquetTableOperations.scala:325, took 20.202567 s
15/01/19 22:54:32 INFO ParquetFileReader: Initiating action with parallelism: 5
parquetSALES: Unit = ()

scala>
```

Querying Parquet files with Impala

Impala is a **massively parallel processing** (**MPP**) data layer that is focused on SQL queries over large data sets and suited for exploratory data analytics. The main utility is the ability of SQL queries over Hadoop data; this means that the data is stored in HDFS in different formats by MapReduce and Spark. Let's fire up Impala and see if we can query our `Orders` database.

The best way to try out Impala is through Cloudera's QuickStart VM available at `http://www.cloudera.com/content/cloudera/en/downloads/quickstart_vms/cdh-5-3-x.html`. While the details are outside the scope of this book, let me quickly outline the top level steps for MacOS:

1. Download the VMWare VM and install via VMWare Fusion.
2. The VM starts and it has a single-node Hadoop cluster with all the stack including Impala, Spark, and HBase, and so on.
3. The VM is CentOS 6.4. You need to start the terminal from **Applications** | **System Tools** | **Terminal**.
4. At the terminal prompt, type `Impala-shell` to start the Impala shell and verify that it works. Type `Exit` and exit out of it.
5. Copy the files under the `Orders.parquet` directory to the `Orders` directory in HDFS in the Cloudera VM using a USB disk. The commands that I used for this are shown here:

   Copy the files to a USB disk:

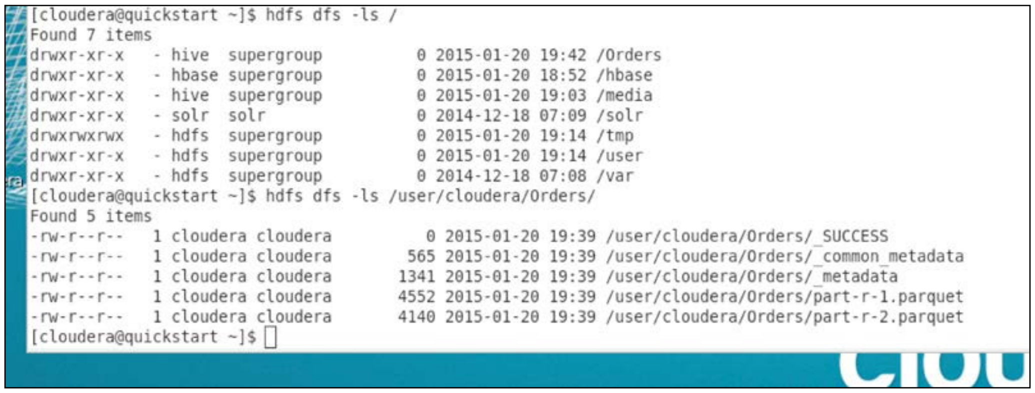
   ```
   cp -rv ~/fdps-vii/Orders.parquet /Volumes/USB\ DISK/
   ```

   Connect the USB to the VM.

   In the VM, copy the files to a local directory first and then to a directory in the HDFS (`hdfsdfs -copyToLocal` gives unexpected `urisyntaxexception` if copied directly from the USB disk—probably the way VMware maps the USB disk), as shown here:

   ```
   [cloudera@quickstart ~]$ mkdir Orders
   [cloudera@quickstart ~]$ cp /media/USB\ DISK/fdps-vii/Orders.
   parquet/* Orders/.
   [cloudera@quickstart ~]$ hdfsdfs -mkdir Orders
   [cloudera@quickstart ~]$ hdfsdfs -copyFromLocal /Orders/* Orders/.
   ```

6. Verify that the files are indeed in HDFS, as shown in the following screenshot:



```
[cloudera@quickstart ~]$ hdfs dfs -ls /
Found 7 items
drwxr-xr-x   - hive  supergroup          0 2015-01-20 19:42 /Orders
drwxr-xr-x   - hbase supergroup          0 2015-01-20 18:52 /hbase
drwxr-xr-x   - hive  supergroup          0 2015-01-20 19:03 /media
drwxr-xr-x   - solr  solr                0 2014-12-18 07:09 /solr
drwxrwxrwx   - hdfs  supergroup          0 2015-01-20 19:14 /tmp
drwxr-xr-x   - hdfs  supergroup          0 2015-01-20 19:14 /user
drwxr-xr-x   - hdfs  supergroup          0 2014-12-18 07:08 /var
[cloudera@quickstart ~]$ hdfs dfs -ls /user/cloudera/Orders/
Found 5 items
-rw-r--r--   1 cloudera cloudera          0 2015-01-20 19:39 /user/cloudera/Orders/_SUCCESS
-rw-r--r--   1 cloudera cloudera        565 2015-01-20 19:39 /user/cloudera/Orders/_common_metadata
-rw-r--r--   1 cloudera cloudera       1341 2015-01-20 19:39 /user/cloudera/Orders/_metadata
-rw-r--r--   1 cloudera cloudera       4552 2015-01-20 19:39 /user/cloudera/Orders/part-r-1.parquet
-rw-r--r--   1 cloudera cloudera       4140 2015-01-20 19:39 /user/cloudera/Orders/part-r-2.parquet
[cloudera@quickstart ~]$
```

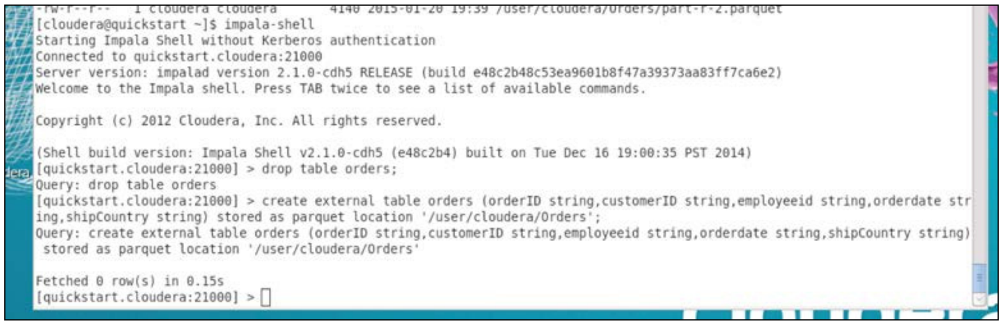
7. Get back to Impala using the following command:

   ```
   [cloudera@quickstart ~]$ impala-shell
   ```

8. Create an external table pointing to the HDFS directory where we have copied the files, as shown here:

   ```
   [quickstart.cloudera:21000] > create external table orders
   (ordered string,customerID string,employeeid string,orderdate
   string,shipCountry string) stored as parquet location '/user/
   cloudera/Orders';
   ```

   The result is shown in the following screenshot:

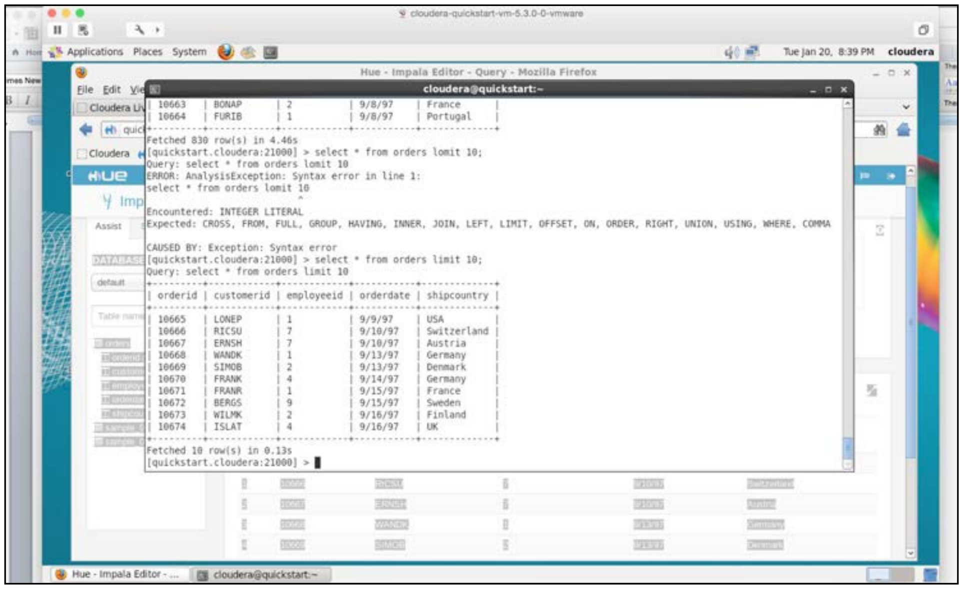


```
[cloudera@quickstart ~]$ impala-shell
Starting Impala Shell without Kerberos authentication
Connected to quickstart.cloudera:21000
Server version: impalad version 2.1.0-cdh5 RELEASE (build e48c2b48c53ea9601b8f47a39373aa83ff7ca6e2)
Welcome to the Impala shell. Press TAB twice to see a list of available commands.

Copyright (c) 2012 Cloudera, Inc. All rights reserved.

(Shell build version: Impala Shell v2.1.0-cdh5 (e48c2b4) built on Tue Dec 16 19:00:35 PST 2014)
[quickstart.cloudera:21000] > drop table orders;
Query: drop table orders
[quickstart.cloudera:21000] > create external table orders (orderID string,customerID string,employeeid string,orderdate str
ing,shipCountry string) stored as parquet location '/user/cloudera/Orders';
Query: create external table orders (orderID string,customerID string,employeeid string,orderdate string,shipCountry string)
 stored as parquet location '/user/cloudera/Orders'

Fetched 0 row(s) in 0.15s
[quickstart.cloudera:21000] >
```

9. Finally, execute the following SQL statement:

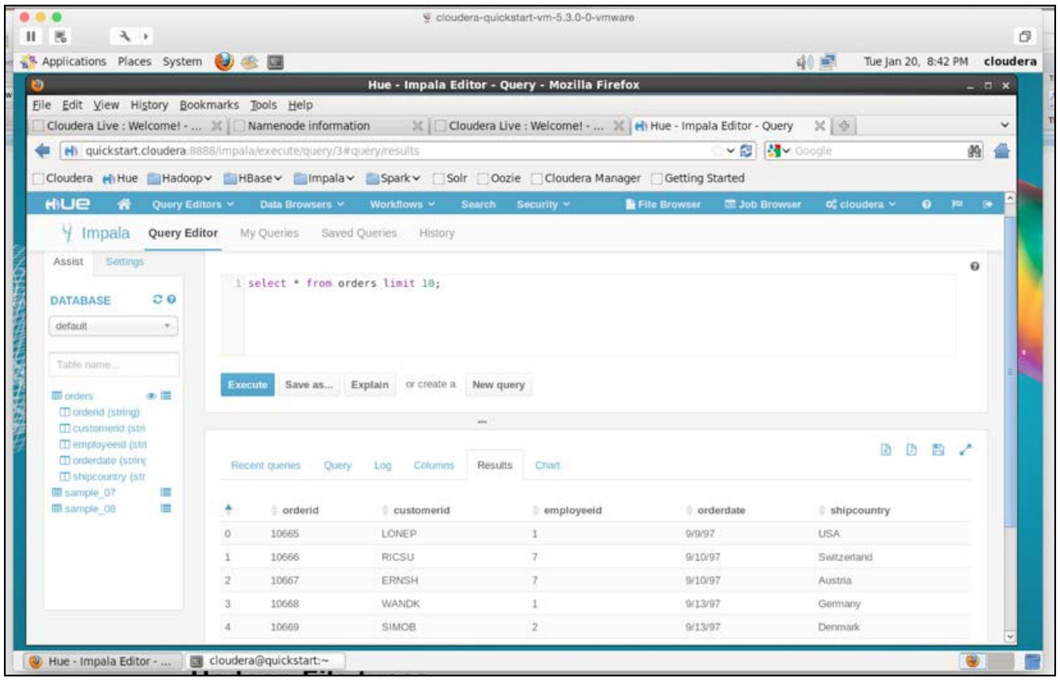
   ```
   select * from orders limit 10;
   ```

   And you can see the records, as shown in the following screenshot:



```
| 10663    | BONAP      | 2          | 9/8/97    | France      |
| 10664    | FURIB      | 1          | 9/8/97    | Portugal    |
+----------+------------+------------+-----------+-------------+
Fetched 830 row(s) in 4.46s
[quickstart.cloudera:21000] > select * from orders lomit 10;
Query: select * from orders lomit 10
ERROR: AnalysisException: Syntax error in line 1:
select * from orders lomit 10
                           ^
Encountered: INTEGER LITERAL
Expected: CROSS, FROM, FULL, GROUP, HAVING, INNER, JOIN, LEFT, LIMIT, OFFSET, ON, ORDER, RIGHT, UNION, USING, WHERE, COMMA

CAUSED BY: Exception: Syntax error
[quickstart.cloudera:21000] > select * from orders limit 10;
Query: select * from orders limit 10
+----------+------------+------------+-----------+-------------+
| orderid | customerid | employeeid | orderdate | shipcountry |
+----------+------------+------------+-----------+-------------+
| 10665    | LONEP      | 1          | 9/9/97    | USA         |
| 10666    | RICSU      | 7          | 9/10/97   | Switzerland |
| 10667    | ERNSH      | 7          | 9/10/97   | Austria     |
| 10668    | WANDK      | 1          | 9/13/97   | Germany     |
| 10669    | SIMOB      | 2          | 9/13/97   | Denmark     |
| 10670    | FRANK      | 4          | 9/14/97   | Germany     |
| 10671    | FRANR      | 1          | 9/15/97   | France      |
| 10672    | BERGS      | 9          | 9/15/97   | Sweden      |
| 10673    | WILMK      | 2          | 9/16/97   | Finland     |
| 10674    | ISLAT      | 4          | 9/16/97   | UK          |
+----------+------------+------------+-----------+-------------+
Fetched 10 row(s) in 0.13s
[quickstart.cloudera:21000] >
```

We can also use the Hue graphical query UI and execute the queries, as shown in the following screenshot:



```
select * from orders limit 10;
```

	orderid	customerid	employeeid	orderdate	shipcountry
0	10665	LONEP	1	9/9/97	USA
1	10666	RICSU	7	9/10/97	Switzerland
2	10667	ERNSH	7	9/10/97	Austria
3	10668	WANDK	1	9/13/97	Germany
4	10669	SIMOB	2	9/13/97	Denmark

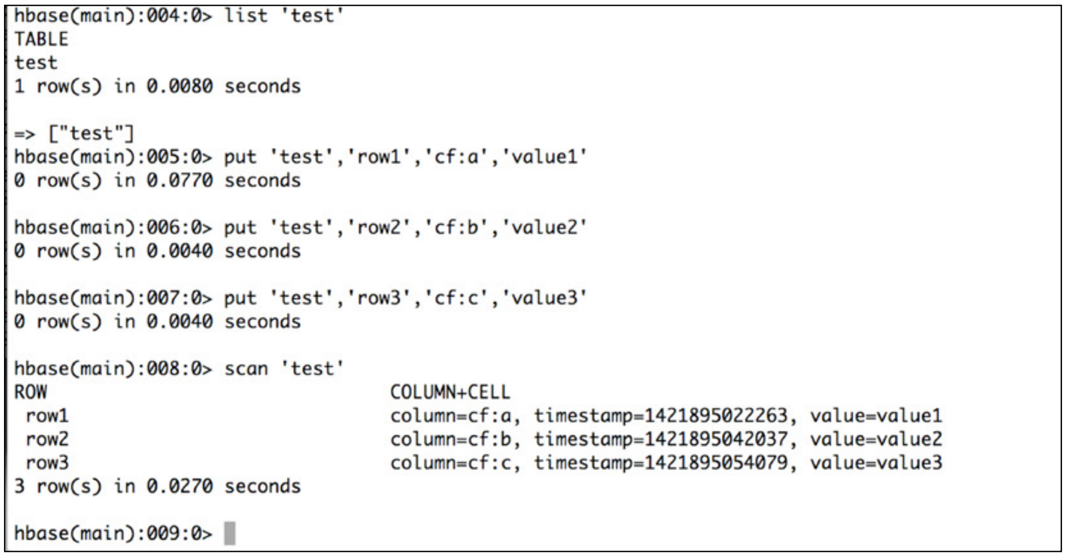
That was not so hard. Once we master the various steps and commands, the rest is easy.

HBase

HBase is the NoSQL datastore in the Hadoop ecosystem. Integration with a database is essential for Spark. It could read data from an HBase table or write to one. In fact, Spark supports HBase very well via the HadoopdataSet calls.

Before working through the examples, let's first create a table and three records in HBase. For testing, you can install a local standalone version of HBase that works from the local filesystem. So there's no need for Hadoop or HDFS. But that won't be suitable for production.

I created a `test` table with three records via the HBase shell as shown in the next screenshot:



```
hbase(main):004:0> list 'test'
TABLE
test
1 row(s) in 0.0080 seconds

=> ["test"]
hbase(main):005:0> put 'test','row1','cf:a','value1'
0 row(s) in 0.0770 seconds

hbase(main):006:0> put 'test','row2','cf:b','value2'
0 row(s) in 0.0040 seconds

hbase(main):007:0> put 'test','row3','cf:c','value3'
0 row(s) in 0.0040 seconds

hbase(main):008:0> scan 'test'
ROW                              COLUMN+CELL
 row1                            column=cf:a, timestamp=1421895022263, value=value1
 row2                            column=cf:b, timestamp=1421895042037, value=value2
 row3                            column=cf:c, timestamp=1421895054079, value=value3
3 row(s) in 0.0270 seconds

hbase(main):009:0> 
```

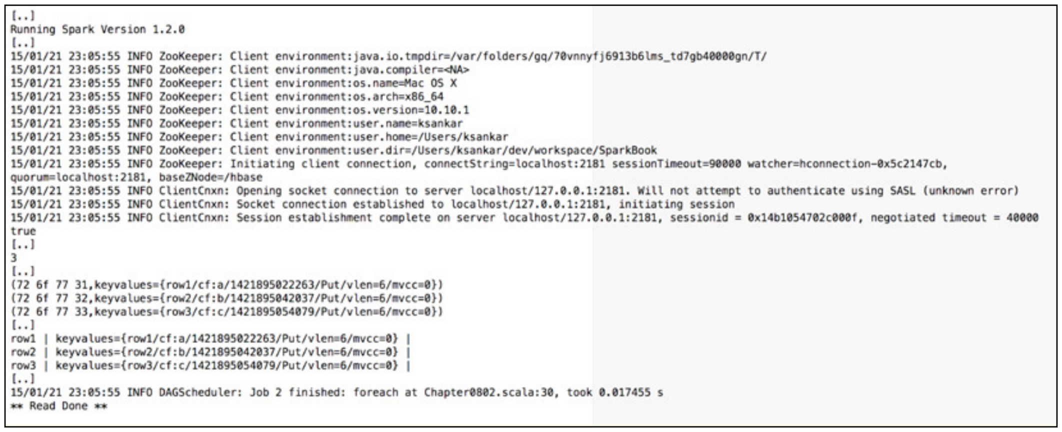
Loading from HBase

The HBase test code in the Apache Spark examples is a good start to test our HBase connectivity and the loading data. The code is not that difficult, but we do need to keep track of the data types, that is, keys as bytes, values as strings, and so on. The test code is given here:

```
Val sc = new SparkContext("local","Chapter 8")
println(s"Running Spark Version ${sc.version}")
//
val conf = HBaseConfiguration.create()
conf.set(TableInputFormat.INPUT_TABLE, "test")

val admin = new HBaseAdmin(conf)
println(admin.isTableAvailable("test"))

val hBaseRDD = sc.newAPIHadoopRDD(conf, classOf[TableInputFormat],
classOf[org.apache.hadoop.hbase.io.ImmutableBytesWritable],
classOf[org.apache.hadoop.hbase.client.Result])
      println(hBaseRDD.count())
      //
      hBaseRDD.foreach(println) // will print bytes
```

```
        hBaseRDD.foreach(e=> ( println("%s | %s |".format(
          Bytes.toString(e._1.get()),e._2) ) ) )
    //
    println("** Read Done **")
```

The output of this is shown in the following screenshot:



```
[..]
Running Spark Version 1.2.0
[..]
15/01/21 23:05:55 INFO ZooKeeper: Client environment:java.io.tmpdir=/var/folders/gq/70vnnyfj6913b6lms_td7gb40000gn/T/
15/01/21 23:05:55 INFO ZooKeeper: Client environment:java.compiler=<NA>
15/01/21 23:05:55 INFO ZooKeeper: Client environment:os.name=Mac OS X
15/01/21 23:05:55 INFO ZooKeeper: Client environment:os.arch=x86_64
15/01/21 23:05:55 INFO ZooKeeper: Client environment:os.version=10.10.1
15/01/21 23:05:55 INFO ZooKeeper: Client environment:user.name=ksankar
15/01/21 23:05:55 INFO ZooKeeper: Client environment:user.home=/Users/ksankar
15/01/21 23:05:55 INFO ZooKeeper: Client environment:user.dir=/Users/ksankar/dev/workspace/SparkBook
15/01/21 23:05:55 INFO ZooKeeper: Initiating client connection, connectString=localhost:2181 sessionTimeout=90000 watcher=hconnection-0x5c2147cb,
quorum=localhost:2181, baseZNode=/hbase
15/01/21 23:05:55 INFO ClientCnxn: Opening socket connection to server localhost/127.0.0.1:2181. Will not attempt to authenticate using SASL (unknown error)
15/01/21 23:05:55 INFO ClientCnxn: Socket connection established to localhost/127.0.0.1:2181, initiating session
15/01/21 23:05:55 INFO ClientCnxn: Session establishment complete on server localhost/127.0.0.1:2181, sessionid = 0x14b1054702c000f, negotiated timeout = 40000
true
[..]
3
[..]
(72 6f 77 31,keyvalues={row1/cf:a/1421895022263/Put/vlen=6/mvcc=0})
(72 6f 77 32,keyvalues={row2/cf:b/1421895042037/Put/vlen=6/mvcc=0})
(72 6f 77 33,keyvalues={row3/cf:c/1421895054079/Put/vlen=6/mvcc=0})
[..]
row1 | keyvalues={row1/cf:a/1421895022263/Put/vlen=6/mvcc=0} |
row2 | keyvalues={row2/cf:b/1421895042037/Put/vlen=6/mvcc=0} |
row3 | keyvalues={row3/cf:c/1421895054079/Put/vlen=6/mvcc=0} |
[..]
15/01/21 23:05:55 INFO DAGScheduler: Job 2 finished: foreach at Chapter0802.scala:30, took 0.017455 s
** Read Done **
```

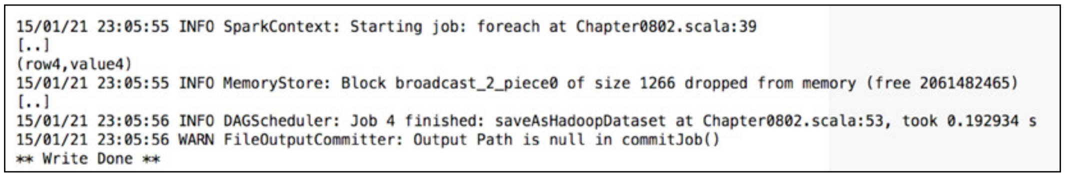
This is just a starting point. You would need to convert the bytes from HBase to the actual data types of your data structures. You need to experiment a bit to get it right.

Saving to HBase

Now let's store a new record in our test table—key as `row4` and value as `value4`. It does require a few more classes and manipulations but nothing fancy, as shown next:

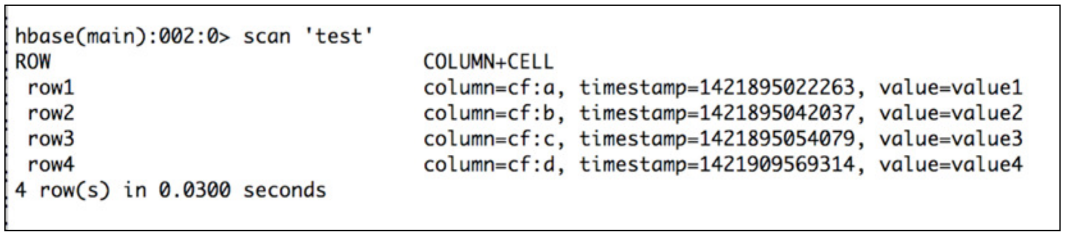
```
        //
        // create a pair RDD "row4":"value4"
        // save it in column family "d"
        //
        val testMap = Map("row4" -> "value4")
        val pairs = sc.parallelize(List(("row4","value4")))
        pairs.foreach(println)
        //
        //Function to convert our RDD to the required format for
          HBase
        //
        def convert(triple: (String, String)) = {
          val p = new Put(Bytes.toBytes(triple._1))
          p.add(Bytes.toBytes("cf"), Bytes.toBytes("d"),
            Bytes.toBytes(triple._2))
          (neworg.apache.hadoop.hbase.io.ImmutableBytesWritable, p)
        }
        //
        valjobConfig: JobConf = new JobConf(conf, this.getClass)
        jobConfig.setOutputFormat(classOf[TableOutputFormat])
        jobConfig.set(TableOutputFormat.OUTPUT_TABLE, "test")
        //
newPairRDDFunctions(pairs.map(convert)).saveAsHadoopDataset(jobCon
  fig)
        //
        println("** Write Done **")
```

The program runs and prints out as shown in the next screenshot:



```
15/01/21 23:05:55 INFO SparkContext: Starting job: foreach at Chapter0802.scala:39
[..]
(row4,value4)
15/01/21 23:05:55 INFO MemoryStore: Block broadcast_2_piece0 of size 1266 dropped from memory (free 2061482465)
[..]
15/01/21 23:05:56 INFO DAGScheduler: Job 4 finished: saveAsHadoopDataset at Chapter0802.scala:53, took 0.192934 s
15/01/21 23:05:56 WARN FileOutputCommitter: Output Path is null in commitJob()
** Write Done **
```

Now let's go back to the HBase shell and verify that the fourth record is added, as shown in the next screenshot:



```
hbase(main):002:0> scan 'test'
ROW                              COLUMN+CELL
 row1                            column=cf:a, timestamp=1421895022263, value=value1
 row2                            column=cf:b, timestamp=1421895042037, value=value2
 row3                            column=cf:c, timestamp=1421895054079, value=value3
 row4                            column=cf:d, timestamp=1421909569314, value=value4
4 row(s) in 0.0300 seconds
```

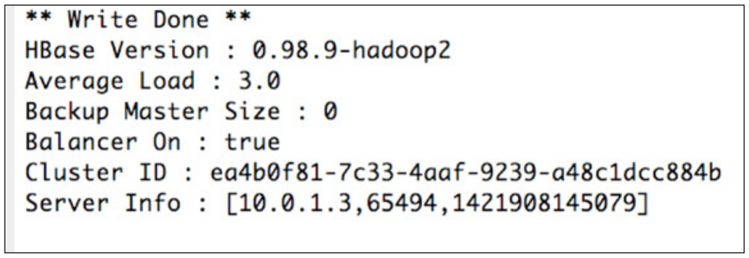
Good. We can see the fourth record and a later timestamp!

Other HBase operations

We can also get the metadata about the HBase server and environment, as shown here:

```
val status = admin.getClusterStatus();
println("HBase Version : " +status.getHBaseVersion())
println("Average Load : "+status.getAverageLoad())
println("Backup Master Size : " + status.getBackupMastersSize())
println("Balancer On : " + status.getBalancerOn())
println("Cluster ID : "+ status.getClusterId())
println("Server Info : " + status.getServerInfo())
```

The output prints out the details, as you can see in the following screenshot:



```
** Write Done **
HBase Version : 0.98.9-hadoop2
Average Load : 3.0
Backup Master Size : 0
Balancer On : true
Cluster ID : ea4b0f81-7c33-4aaf-9239-a48c1dcc884b
Server Info : [10.0.1.3,65494,1421908145079]
```

Some more information is available at the following websites:

- `https://github.com/apache/spark/tree/master/examples/src/main/scala/org/apache/spark/examples`
- `http://parquet.incubator.apache.org/documentation/latest/`
- `http://www.slideshare.net/cloudera/hadoop-summit-36479635?ref=http://parquet.incubator.apache.org/presentations/`
- Google Dremel paper at `http://research.google.com/pubs/pub36632.html`
- `https://blog.twitter.com/2013/dremel-made-simple-with-parquet`
- `http://planetcassandra.org/getting-started-with-apache-spark-and-cassandra/`
- `http://blog.cloudera.com/blog/2014/12/new-in-cloudera-labs-sparkonhbase/`
- `http://www.vidyasource.com/blog/Programming/Scala/Java/Data/Hadoop/Analytics/2014/01/25/lighting-a-spark-with-hbase`
- `https://github.com/apache/spark/blob/master/examples/src/main/scala/org/apache/spark/examples/HBaseTest.scala`
- `https://federicodayan.wordpress.com/2010/09/28/hbase-textgetbytes-and-immutablebyteswritabletostring/`

Summary

This chapter was focused on the integration of Spark with other big data technologies. The Parquet format is an excellent way to expose the data processed by Spark to external systems, and Impala makes this very easy. The advantage of the Parquet format is that it is very efficient in terms of storage and expressive enough to capture the schema. We also looked at the process of interfacing with HBase. Thus, we can have our cake and eat it too! This means that we can leverage Spark for distributed scalable data processing, without losing the capability to integrate with other big data technologies.

9
Machine Learning Using Spark MLlib

One of the major attractions of Spark is the ability to scale computation massively, and that is exactly what you need for machine learning algorithms. But the caveat is that all machine learning algorithms cannot be effectively parallelized. Each algorithm has its own challenges for parallelization, whether it is task parallelism or data parallelism. Having said that, Spark is becoming the de-facto platform for building machine learning algorithms and applications. For example, Apache Mahout is moving away from Hadoop MapReduce and implementing the algorithms in Spark (see the first reference at the end of this chapter). The developers working on the Spark MLlib are implementing more and more machine algorithms in a scalable and concise manner in the Spark framework. For the latest information on this, you can refer to the Spark site at `https://spark.apache.org/docs/latest/mllib-guide.html`, which is the authoritative source.

This chapter covers the following machine learning algorithms:

- Basic statistics
- Linear regression
- Classification
- Clustering
- Recommendations

The Spark machine learning algorithm table

The Spark machine learning algorithms implemented in Spark 1.1.0 `org.apache.spark.mllib` for Scala and Java, and in `pyspark.mllib` for Python is shown in the following table:

Algorithm	**Feature**	**Notes**
Basic statistics	Summary statistics	Mean, variance, count, max, min, and numNonZeros
	Correlations	Spearman and Pearson correlation
	Stratified sampling	sampleBykey, sampleByKeyExact—With and without replacement
	Hypothesis testing	Pearson's chi-squared goodness of fit test
	Random data generation	RandomRDDs Normal, Poisson, and so on
Regression	Linear models	Linear regression—least square, Lasso, and ridge regression
Classification	Binary classification	Logistic regression, SVM, decision trees, and naïve Bayes
	Multi-class classification	Decision trees, naïve Bayes, and so on
Recommendation	Collaborative filtering	Alternating least squares
Clustering	k-means	
Dimensionality reduction	SVD PCA	
Feature extraction	TF-IDF Word2Vec StandardScaler Normalizer	
Optimization	SGD L-BFGS	

Spark MLlib examples

Now, let's look at how to use the algorithms. Naturally, we need interesting datasets to implement the algorithms; we will use appropriate datasets for the algorithms shown in the next section. In the book text, we will use Scala, but I have included iPython notebooks of the algorithm examples in Python as well.

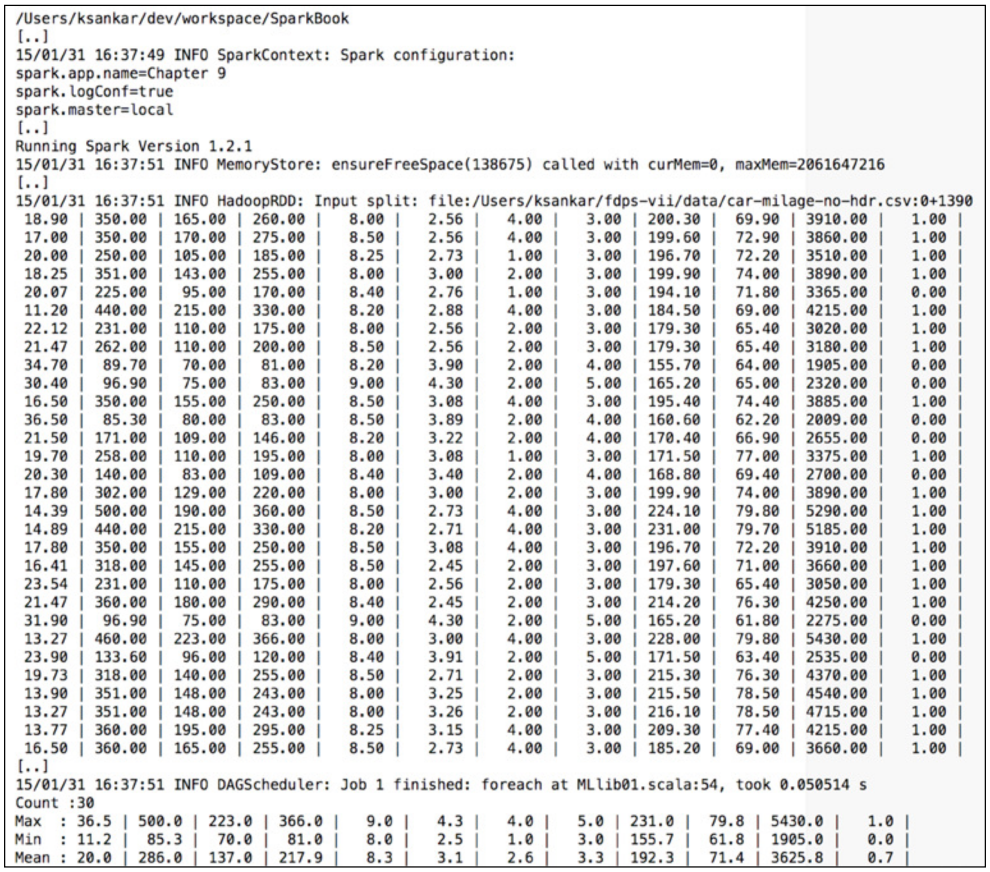
The code and data files are available in the GitHub repository at `https://github.com/xsankar/fdps-vii`. We'll keep it updated with corrections.

Basic statistics

Let's read the car mileage data into an RDD and then compute some basic statistics. We will use a simple parse class to parse a line of data. This will work if you know the type and the structure of your CSV file. We will use this technique for the examples in this chapter:

```
import org.apache.spark.SparkContext
import org.apache.spark.mllib.stat.
  {MultivariateStatisticalSummary, Statistics}
import org.apache.spark.mllib.linalg.Vector
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.rdd.RDD

object MLlib01 {
  //
  def getCurrentDirectory = new java.io.File( "." ).getCanonicalPath
  //
  def parseCarData(inpLine : String) : Array[Double] = {
    val values = inpLine.split(',')
    val mpg = values(0).toDouble
    val displacement = values(1).toDouble
    val hp = values(2).toInt
    val torque = values(3).toInt
    val CRatio = values(4).toDouble
    val RARatio = values(5).toDouble
    val CarbBarrells = values(6).toInt
    val NoOfSpeed = values(7).toInt
    val length = values(8).toDouble
    val width = values(9).toDouble
    val weight = values(10).toDouble
    val automatic = values(11).toInt
    return Array(mpg,displacement,hp,
    torque,CRatio,RARatio,CarbBarrells,
    NoOfSpeed,length,width,weight,automatic)
  }
  //
  def main(args: Array[String]) {
    println(getCurrentDirectory)
    val sc = new SparkContext("local","Chapter 9")
    println(s"Running Spark Version ${sc.version}")
    //
   val dataFile = sc.textFile("/Users/ksankar/fdps-vii/data/car-
     milage-no-hdr.csv")
   val carRDD = dataFile.map(line => parseCarData(line))
   //
   // Let us find summary statistics
   //
   val vectors: RDD[Vector] = carRDD.map(v => Vectors.dense(v))
   val summary = Statistics.colStats(vectors)
   carRDD.foreach(ln=> {ln.foreach(no => print("%6.2f | "
     .format(no))); println()})
   print("Max  :");summary.max.toArray.foreach(m => print("%5.1f |
     ".format(m)));println
   print("Min  :");summary.min.toArray.foreach(m => print("%5.1f |
     ".format(m)));println
   print("Mean :");summary.mean.toArray.foreach(m => print("%5.1f
     | ".format(m)));println
   }
}
```

This program will produce the following output:

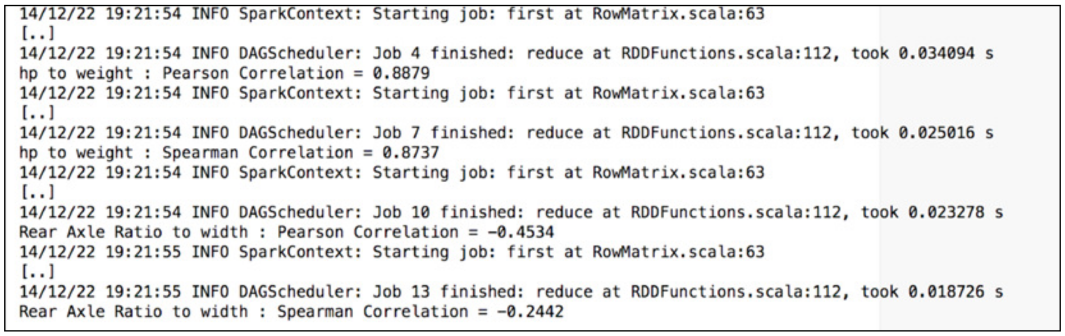


```
/Users/ksankar/dev/workspace/SparkBook
[..]
15/01/31 16:37:49 INFO SparkContext: Spark configuration:
spark.app.name=Chapter 9
spark.logConf=true
spark.master=local
[..]
Running Spark Version 1.2.1
15/01/31 16:37:51 INFO MemoryStore: ensureFreeSpace(138675) called with curMem=0, maxMem=2061647216
[..]
15/01/31 16:37:51 INFO HadoopRDD: Input split: file:/Users/ksankar/fdps-vii/data/car-milage-no-hdr.csv:0+1390
 18.90 | 350.00 | 165.00 | 260.00 |   8.00 |   2.56 |   4.00 |   3.00 | 200.30 |  69.90 | 3910.00 |   1.00 |
 17.00 | 350.00 | 170.00 | 275.00 |   8.50 |   2.56 |   4.00 |   3.00 | 199.60 |  72.90 | 3860.00 |   1.00 |
 20.00 | 250.00 | 105.00 | 185.00 |   8.25 |   2.73 |   1.00 |   3.00 | 196.70 |  72.20 | 3510.00 |   1.00 |
 18.25 | 351.00 | 143.00 | 255.00 |   8.00 |   3.00 |   2.00 |   3.00 | 199.90 |  74.00 | 3890.00 |   1.00 |
 20.07 | 225.00 |  95.00 | 170.00 |   8.40 |   2.76 |   1.00 |   3.00 | 194.10 |  71.80 | 3365.00 |   0.00 |
 11.20 | 440.00 | 215.00 | 330.00 |   8.20 |   2.88 |   4.00 |   3.00 | 184.50 |  69.00 | 4215.00 |   1.00 |
 22.12 | 231.00 | 110.00 | 175.00 |   8.00 |   2.56 |   2.00 |   3.00 | 179.30 |  65.40 | 3020.00 |   1.00 |
 21.47 | 262.00 | 110.00 | 200.00 |   8.50 |   2.56 |   2.00 |   3.00 | 179.30 |  65.40 | 3180.00 |   1.00 |
 34.70 |  89.70 |  70.00 |  81.00 |   8.20 |   3.90 |   2.00 |   4.00 | 155.70 |  64.00 | 1905.00 |   0.00 |
 30.40 |  96.90 |  75.00 |  83.00 |   9.00 |   4.30 |   2.00 |   5.00 | 165.20 |  65.00 | 2320.00 |   0.00 |
 16.50 | 350.00 | 155.00 | 250.00 |   8.50 |   3.08 |   4.00 |   3.00 | 195.40 |  74.40 | 3885.00 |   1.00 |
 36.50 |  85.30 |  80.00 |  83.00 |   8.50 |   3.89 |   2.00 |   4.00 | 160.60 |  62.20 | 2009.00 |   0.00 |
 21.50 | 171.00 | 109.00 | 146.00 |   8.20 |   3.22 |   2.00 |   4.00 | 170.40 |  66.90 | 2655.00 |   0.00 |
 19.70 | 258.00 | 110.00 | 195.00 |   8.00 |   3.08 |   1.00 |   3.00 | 171.50 |  77.00 | 3375.00 |   1.00 |
 20.30 | 140.00 |  83.00 | 109.00 |   8.40 |   3.40 |   2.00 |   4.00 | 168.80 |  69.40 | 2700.00 |   0.00 |
 17.80 | 302.00 | 129.00 | 220.00 |   8.00 |   3.00 |   2.00 |   3.00 | 199.90 |  74.00 | 3890.00 |   1.00 |
 14.39 | 500.00 | 190.00 | 360.00 |   8.50 |   2.73 |   4.00 |   3.00 | 224.10 |  79.80 | 5290.00 |   1.00 |
 14.89 | 440.00 | 215.00 | 330.00 |   8.20 |   2.71 |   4.00 |   3.00 | 231.00 |  79.70 | 5185.00 |   1.00 |
 17.80 | 350.00 | 155.00 | 250.00 |   8.50 |   3.08 |   4.00 |   3.00 | 196.70 |  72.20 | 3910.00 |   1.00 |
 16.41 | 318.00 | 145.00 | 255.00 |   8.50 |   2.45 |   2.00 |   3.00 | 197.60 |  71.00 | 3660.00 |   1.00 |
 23.54 | 231.00 | 110.00 | 175.00 |   8.00 |   2.56 |   2.00 |   3.00 | 179.30 |  65.40 | 3050.00 |   1.00 |
 21.47 | 360.00 | 180.00 | 290.00 |   8.40 |   2.45 |   2.00 |   3.00 | 214.20 |  76.30 | 4250.00 |   1.00 |
 31.90 |  96.90 |  75.00 |  83.00 |   9.00 |   4.30 |   2.00 |   5.00 | 165.20 |  61.80 | 2275.00 |   0.00 |
 13.27 | 460.00 | 223.00 | 366.00 |   8.00 |   3.00 |   4.00 |   3.00 | 228.00 |  79.80 | 5430.00 |   1.00 |
 23.90 | 133.60 |  96.00 | 120.00 |   8.40 |   3.91 |   2.00 |   5.00 | 171.50 |  63.40 | 2535.00 |   0.00 |
 19.73 | 318.00 | 140.00 | 255.00 |   8.50 |   2.71 |   2.00 |   3.00 | 215.30 |  76.30 | 4370.00 |   1.00 |
 13.90 | 351.00 | 148.00 | 243.00 |   8.00 |   3.25 |   2.00 |   3.00 | 215.50 |  78.50 | 4540.00 |   1.00 |
 13.27 | 351.00 | 148.00 | 243.00 |   8.00 |   3.26 |   2.00 |   3.00 | 216.10 |  78.50 | 4715.00 |   1.00 |
 13.77 | 360.00 | 195.00 | 295.00 |   8.25 |   3.15 |   4.00 |   3.00 | 209.30 |  77.40 | 4215.00 |   1.00 |
 16.50 | 360.00 | 165.00 | 255.00 |   8.50 |   2.73 |   4.00 |   3.00 | 185.20 |  69.00 | 3660.00 |   1.00 |
[..]
15/01/31 16:37:51 INFO DAGScheduler: Job 1 finished: foreach at MLlib01.scala:54, took 0.050514 s
Count :30
Max  : 36.5 | 500.0 | 223.0 | 366.0 |   9.0 |   4.3 |   4.0 |   5.0 | 231.0 |  79.8 | 5430.0 |   1.0 |
Min  : 11.2 |  85.3 |  70.0 |  81.0 |   8.0 |   2.5 |   1.0 |   3.0 | 155.7 |  61.8 | 1905.0 |   0.0 |
Mean : 20.0 | 286.0 | 137.0 | 217.9 |   8.3 |   3.1 |   2.6 |   3.3 | 192.3 |  71.4 | 3625.8 |   0.7 |
```

Let's also run some correlations, as shown here:

```
//
// correlations
//
val hp = vectors.map(x => x(2))
val weight = vectors.map(x => x(10))
var corP = Statistics.corr(hp,weight,"pearson") // default
println("hp to weight : Pearson Correlation = %2.4f".format(corP))
var corS = Statistics.corr(hp,weight,"spearman") // Need to
  specify
println("hp to weight : Spearman Correlation = %2.4f"
  .format(corS))
//
val raRatio = vectors.map(x => x(5))
val width = vectors.map(x => x(9))
corP = Statistics.corr(raRatio,width,"pearson") // default
println("raRatio to width : Pearson Correlation = %2.4f"
  .format(corP))
corS = Statistics.corr(raRatio,width,"spearman") // Need to
  specify
println("raRatio to width : Spearman Correlation = %2.4f"
  .format(corS))
//
```

This will produce interesting results as shown in the next screenshot:



```
14/12/22 19:21:54 INFO SparkContext: Starting job: first at RowMatrix.scala:63
[..]
14/12/22 19:21:54 INFO DAGScheduler: Job 4 finished: reduce at RDDFunctions.scala:112, took 0.034094 s
hp to weight : Pearson Correlation = 0.8879
14/12/22 19:21:54 INFO SparkContext: Starting job: first at RowMatrix.scala:63
[..]
14/12/22 19:21:54 INFO DAGScheduler: Job 7 finished: reduce at RDDFunctions.scala:112, took 0.025016 s
hp to weight : Spearman Correlation = 0.8737
14/12/22 19:21:54 INFO SparkContext: Starting job: first at RowMatrix.scala:63
[..]
14/12/22 19:21:54 INFO DAGScheduler: Job 10 finished: reduce at RDDFunctions.scala:112, took 0.023278 s
Rear Axle Ratio to width : Pearson Correlation = -0.4534
14/12/22 19:21:55 INFO SparkContext: Starting job: first at RowMatrix.scala:63
[..]
14/12/22 19:21:55 INFO DAGScheduler: Job 13 finished: reduce at RDDFunctions.scala:112, took 0.018726 s
Rear Axle Ratio to width : Spearman Correlation = -0.2442
```

While this might seem too much work to calculate the correlation of a tiny dataset, remember that this will scale to datasets consisting of 1,000,000 rows or even a billion rows!

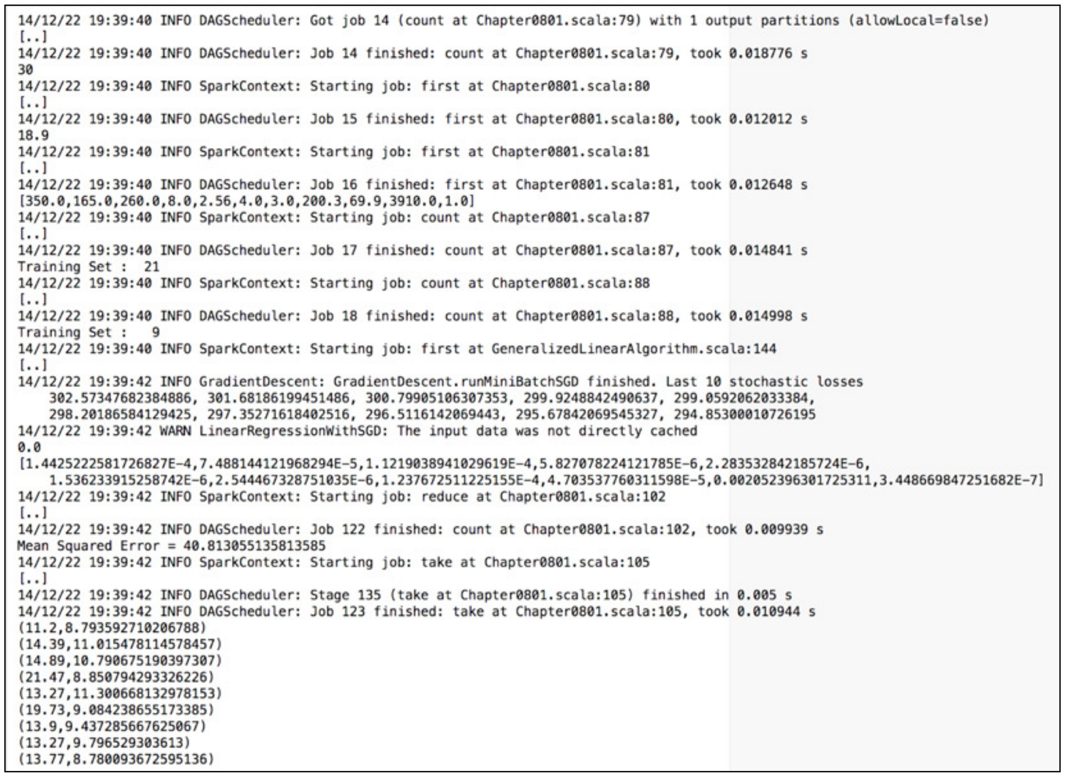
Linear regression

Linear regression takes a little more work than statistics. We need the `LabeledPoint` class as well as a few more parameters such as the learning rate, that is, the step size. We will also split the dataset into `training` and `test`, as shown here:

```
      //
      //
     def carDataToLP(inpArray : Array[Double]) : LabeledPoint = {
       return new LabeledPoint( inpArray(0),Vectors.dense (
         inpArray(1), inpArray(2), inpArray(3),
         inpArray(4), inpArray(5), inpArray(6), inpArray(7),
         inpArray(8), inpArray(9), inpArray(10), inpArray(11) ) )
       }
   // Linear Regression
      //
      val carRDDLP = carRDD.map(x => carDataToLP(x)) // create a
        labeled point RDD
      println(carRDDLP.count())
      println(carRDDLP.first().label)
      println(carRDDLP.first().features)
      //
      // Let us split the data set into training & test set using a
        very simple filter
      //
      val carRDDLPTrain = carRDDLP.filter( x => x.features(9) <=
        4000)
      val carRDDLPTest = carRDDLP.filter( x => x.features(9) > 4000)
      println("Training Set : " + "%3d".format
        (carRDDLPTrain.count()))
      println("Training Set : " + "%3d".format(carRDDLPTest.count()))
      //
      // Train a Linear Regression Model
      // numIterations = 100, stepsize = 0.000000001
      // without such a small step size the algorithm will diverge
      //
      val mdlLR = LinearRegressionWithSGD.train
        (carRDDLPTrain,100,0.000000001)
      println(mdlLR.intercept) // Intercept is turned off when using
        LinearRegressionSGD object, so intercept will always be 0 for
        this code

      println(mdlLR.weights)
      //
      // Now let us use the model to predict our test set
      //
      val valuesAndPreds = carRDDLPTest.map(p => (p.label,
        mdlLR.predict(p.features)))
```

```
val mse = valuesAndPreds.map( vp => math.pow( (vp._1 - vp._2),2
  ) ).
    reduce(_+_) / valuesAndPreds.count()
println("Mean Squared Error      = " + "%6.3f".format(mse))
 println("Root Mean Squared Error = " + "%6.3f"
   .format(math.sqrt(mse)))
 // Let us print what the model predicted
 valuesAndPreds.take(20).foreach(m => println("%5.1f | %5.1f |"
   .format(m._1,m._2)))
```

The run result will be as expected, as shown in the next screenshot:



```
14/12/22 19:39:40 INFO DAGScheduler: Got job 14 (count at Chapter0801.scala:79) with 1 output partitions (allowLocal=false)
[..]
14/12/22 19:39:40 INFO DAGScheduler: Job 14 finished: count at Chapter0801.scala:79, took 0.018776 s
30
14/12/22 19:39:40 INFO SparkContext: Starting job: first at Chapter0801.scala:80
[..]
14/12/22 19:39:40 INFO DAGScheduler: Job 15 finished: first at Chapter0801.scala:80, took 0.012012 s
18.9
14/12/22 19:39:40 INFO SparkContext: Starting job: first at Chapter0801.scala:81
[..]
14/12/22 19:39:40 INFO DAGScheduler: Job 16 finished: first at Chapter0801.scala:81, took 0.012648 s
[350.0,165.0,260.0,8.0,2.56,4.0,3.0,200.3,69.9,3910.0,1.0]
14/12/22 19:39:40 INFO SparkContext: Starting job: count at Chapter0801.scala:87
[..]
14/12/22 19:39:40 INFO DAGScheduler: Job 17 finished: count at Chapter0801.scala:87, took 0.014841 s
Training Set :  21
14/12/22 19:39:40 INFO SparkContext: Starting job: count at Chapter0801.scala:88
[..]
14/12/22 19:39:40 INFO DAGScheduler: Job 18 finished: count at Chapter0801.scala:88, took 0.014998 s
Training Set :   9
14/12/22 19:39:40 INFO SparkContext: Starting job: first at GeneralizedLinearAlgorithm.scala:144
[..]
14/12/22 19:39:42 INFO GradientDescent: GradientDescent.runMiniBatchSGD finished. Last 10 stochastic losses
    302.57347682384886, 301.68186199451486, 300.79905106307353, 299.9248842490637, 299.0592062033384,
    298.20186584129425, 297.35271618402516, 296.5116142069443, 295.67842069545327, 294.85300010726195
14/12/22 19:39:42 WARN LinearRegressionWithSGD: The input data was not directly cached
0.0
[1.4425222581726827E-4,7.488144121968294E-5,1.1219038941029619E-4,5.827078224121785E-6,2.283532842185724E-6,
    1.536233915258742E-6,2.544467328751035E-6,1.237672511225155E-4,4.703537760311598E-5,0.002052396301725311,3.448669847251682E-7]
14/12/22 19:39:42 INFO SparkContext: Starting job: reduce at Chapter0801.scala:102
[..]
14/12/22 19:39:42 INFO DAGScheduler: Job 122 finished: count at Chapter0801.scala:102, took 0.009939 s
Mean Squared Error = 40.813055135813585
14/12/22 19:39:42 INFO SparkContext: Starting job: take at Chapter0801.scala:105
[..]
14/12/22 19:39:42 INFO DAGScheduler: Stage 135 (take at Chapter0801.scala:105) finished in 0.005 s
14/12/22 19:39:42 INFO DAGScheduler: Job 123 finished: take at Chapter0801.scala:105, took 0.010944 s
(11.2,8.793592710206788)
(14.39,11.015478114578457)
(14.89,10.790675190397307)
(21.47,8.850794293326226)
(13.27,11.300668132978153)
(19.73,9.084238655173385)
(13.9,9.437285667625067)
(13.27,9.796529303613)
(13.77,8.780093672595136)
```

The prediction is not that impressive. There are a couple of reasons for this. There might be quadratic effects; some of the variables might be correlated (for example, length, width, and weight, and so we might not need all three to predict the `mpg` value). Finally, we might not need all the 10 features anyways. I leave it to you to try with different combinations of features. (In the `parseCarData` function, take only a subset of the variables; for example take hp, weight, and number of speed and see which combination minimizes the `mse` value.)

Classification

Classification is very similar to linear regression. The algorithms take labeled points, and the train process has various parameters to tweak the algorithm to fit the needs of an application. The returned model can be used to predict the class of a labeled point. Here is a quick example using the `titanic` dataset:

For our example, we will keep the same structure as the linear regression example. First, we will parse the full dataset line and then later keep it simple by creating a labeled point with a set of selected features, as shown in the following code:

```
import org.apache.spark.SparkContext
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.tree.DecisionTree

object Chapter0802 {
  //
  def getCurrentDirectory = new java.io.File( "."
    ).getCanonicalPath
  //
  //  0 pclass,1 survived,2 l.name,3.f.name, 4 sex,5 age,6 sibsp,7
      parch,8 ticket,9 fare,10 cabin,
  // 11 embarked,12 boat,13 body,14 home.dest
  //
  def str2Double(x: String) : Double = {
    try {
      x.toDouble
    } catch {
      case e: Exception => 0.0
    }
  }
  //
  def parsePassengerDataToLP(inpLine : String) : LabeledPoint = {
    val values = inpLine.split(',')
    //println(values)
    //println(values.length)
    //
    val pclass = str2Double(values(0))
    val survived = str2Double(values(1))
    // skip last name, first name
    var sex = 0
    if (values(4) == "male") {
      sex = 1
    }
```

```
        var age = 0.0 // a better choice would be the average of all
          ages
        age = str2Double(values(5))
        //
        var sibsp = 0.0
        age = str2Double(values(6))
        //
        var parch = 0.0
        age = str2Double(values(7))
        //
        var fare = 0.0
        fare = str2Double(values(9))
        return new LabeledPoint(survived,Vectors.dense
          (pclass,sex,age,sibsp,parch,fare))
      }
```

Now that we have setup the routines to parse the data, let's dive into the main program:

```
      //
      def main(args: Array[String]): Unit = {
        println(getCurrentDirectory)
        val sc = new SparkContext("local","Chapter 8")
        println(s"Running Spark Version ${sc.version}")
        //
        val dataFile = sc.textFile("/Users/ksankar/bdtc-2014
          /titanic/titanic3_01.csv")
        val titanicRDDLP = dataFile.map(_.trim).filter( _.length > 1).
          map(line => parsePassengerDataToLP(line))
        //
        println(titanicRDDLP.count())
        //titanicRDDLP.foreach(println)
        //
        println(titanicRDDLP.first().label)
        println(titanicRDDLP.first().features)
        //
        val categoricalFeaturesInfo = Map[Int, Int]()
        val mdlTree = DecisionTree.trainClassifier(titanicRDDLP, 2, //
          numClasses
            categoricalFeaturesInfo, // all features are continuous
            "gini", // impurity
            5, // Maxdepth
            32) //maxBins
        //
        println(mdlTree.depth)
        println(mdlTree)
```

The tree is interesting to inspect. Check it out here:

```
    //
    // Let us predict on the dataset and see how well it works.
    // In the real world, we should split the data to train & test
       and then predict the test data:
    //
    val predictions = mdlTree.predict(titanicRDDLP.
      map(x=>x.features))
    val labelsAndPreds = titanicRDDLP.
      map(x=>x.label).zip(predictions)
    //
    val mse = labelsAndPreds.map( vp => math.pow( (vp._1 -
      vp._2),2 ) ).
       reduce(_+_) / labelsAndPreds.count()
    println("Mean Squared Error = " + "%6f".format(mse))
    //
    // labelsAndPreds.foreach(println)
    //
    val correctVals = labelsAndPreds.aggregate(0.0)((x, rec) => x
      + (rec._1 == rec._2).compare(false), _ + _)
    val accuracy = correctVals/labelsAndPreds.count()
    println("Accuracy = " + "%3.2f%%".format(accuracy*100))
    //
    println("*** Done ***")
  }
}
```

The result obtained when you run the program is as expected. The printout of the tree is interesting, as shown here:

```
Running Spark Version 1.1.1
14/11/28 18:41:27 INFO MemoryStore: ensureFreeSpace(163705) called with
curMem=0, maxMem=2061647216
[..]
14/11/28 18:41:27 INFO SparkContext: Job finished: count at Chapter0802.
scala:56, took 0.260993 s
1309
14/11/28 18:41:27 INFO SparkContext: Starting job: first at Chapter0802.
scala:59
[..]
14/11/28 18:41:27 INFO SparkContext: Job finished: first at Chapter0802.
scala:59, took 0.016479 s
1.0
```

```
14/11/28 18:41:27 INFO SparkContext: Starting job: first at Chapter0802.
scala:60
[..]
14/11/28 18:41:27 INFO SparkContext: Job finished: first at Chapter0802.
scala:60, took 0.014408 s
[1.0,0.0,0.0,0.0,0.0,211.3375]
14/11/28 18:41:27 INFO SparkContext: Starting job: take at
DecisionTreeMetadata.scala:66
[..]
14/11/28 18:41:28 INFO DecisionTree: Internal timing for DecisionTree:
14/11/28 18:41:28 INFO DecisionTree:   init: 0.36408
  total: 0.95518
  extractNodeInfo: 7.3E-4
  findSplitsBins: 0.249814
  extractInfoForLowerLevels: 7.74E-4
  findBestSplits: 0.565394
  chooseSplits: 0.201012
  aggregation: 0.362411
5
DecisionTreeModel classifier
  If (feature 1 <= 0.0)
   If (feature 0 <= 2.0)
    If (feature 5 <= 26.0)
     If (feature 2 <= 1.0)
      If (feature 0 <= 1.0)
       Predict: 1.0
      Else (feature 0 > 1.0)
       Predict: 1.0
     Else (feature 2 > 1.0)
      Predict: 1.0
    Else (feature 5 > 26.0)
     If (feature 2 <= 1.0)
      If (feature 5 <= 38.0021)
       Predict: 1.0
      Else (feature 5 > 38.0021)
       Predict: 1.0
     Else (feature 2 > 1.0)
      If (feature 5 <= 79.42500000000001)
       Predict: 1.0
      Else (feature 5 > 79.42500000000001)
```

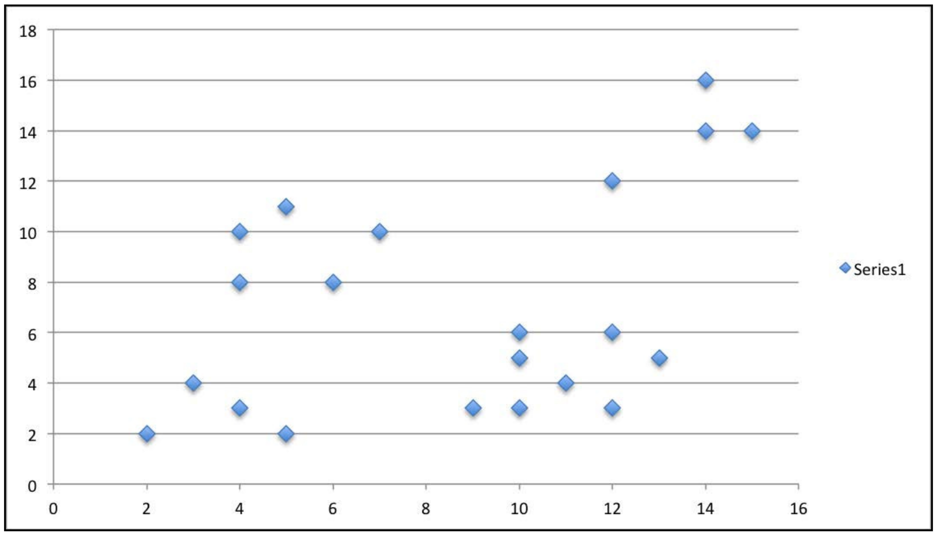
```
       Predict: 1.0
   Else (feature 0 > 2.0)
    If (feature 5 <= 25.4667)
     If (feature 5 <= 7.2292)
      If (feature 5 <= 7.05)
       Predict: 1.0
      Else (feature 5 > 7.05)
       Predict: 1.0
     Else (feature 5 > 7.2292)
      If (feature 5 <= 15.5646)
       Predict: 0.0
      Else (feature 5 > 15.5646)
       Predict: 1.0
    Else (feature 5 > 25.4667)
     If (feature 5 <= 38.0021)
      If (feature 5 <= 30.6958)
       Predict: 0.0
      Else (feature 5 > 30.6958)
       Predict: 0.0
     Else (feature 5 > 38.0021)
      Predict: 0.0
  Else (feature 1 > 0.0)
   If (feature 0 <= 1.0)
    If (feature 5 <= 26.0)
     If (feature 5 <= 7.05)
      If (feature 5 <= 0.0)
       Predict: 0.0
      Else (feature 5 > 0.0)
       Predict: 0.0
     Else (feature 5 > 7.05)
      Predict: 0.0
    Else (feature 5 > 26.0)
     If (feature 5 <= 30.6958)
      If (feature 2 <= 0.0)
       Predict: 0.0
      Else (feature 2 > 0.0)
       Predict: 0.0
     Else (feature 5 > 30.6958)
      If (feature 2 <= 1.0)
       Predict: 0.0
      Else (feature 2 > 1.0)
       Predict: 1.0
   Else (feature 0 > 1.0)
    If (feature 2 <= 0.0)
     If (feature 5 <= 38.0021)
      If (feature 5 <= 14.4583)
       Predict: 0.0
      Else (feature 5 > 14.4583)
       Predict: 0.0
     Else (feature 5 > 38.0021)
      If (feature 0 <= 2.0)
       Predict: 0.0
      Else (feature 0 > 2.0)
       Predict: 1.0
    Else (feature 2 > 0.0)
     If (feature 5 <= 26.0)
      If (feature 2 <= 1.0)
       Predict: 0.0
      Else (feature 2 > 1.0)
       Predict: 0.0
     Else (feature 5 > 26.0)
      If (feature 0 <= 2.0)
       Predict: 0.0
      Else (feature 0 > 2.0)
       Predict: 0.0

14/11/28 18:41:28 INFO SparkContext: Starting job: reduce at Chapter0802.
scala:79
[..]
14/11/28 18:41:28 INFO SparkContext: Job finished: count at Chapter0802.
scala:79, took 0.077973 s
Mean Squared Error = 0.200153
14/11/28 18:41:28 INFO SparkContext: Starting job: aggregate at
Chapter0802.scala:84
[..]
14/11/28 18:41:28 INFO SparkContext: Job finished: count at Chapter0802.
scala:85, took 0.042592 s
Accuracy = 79.98%
*** Done ***
```

In the real world, one would create a `training` and a `test` dataset and train the model on the `training` dataset and then predict on the `test` dataset. Then we can calculate the `mse` and minimize it on various feature combinations, some of which could also be engineered features.

Clustering

Spark MLlib has implemented the k-means clustering algorithm. The model training and prediction interfaces are similar to other machine learning algorithms. Let's see how it works by going through an example.

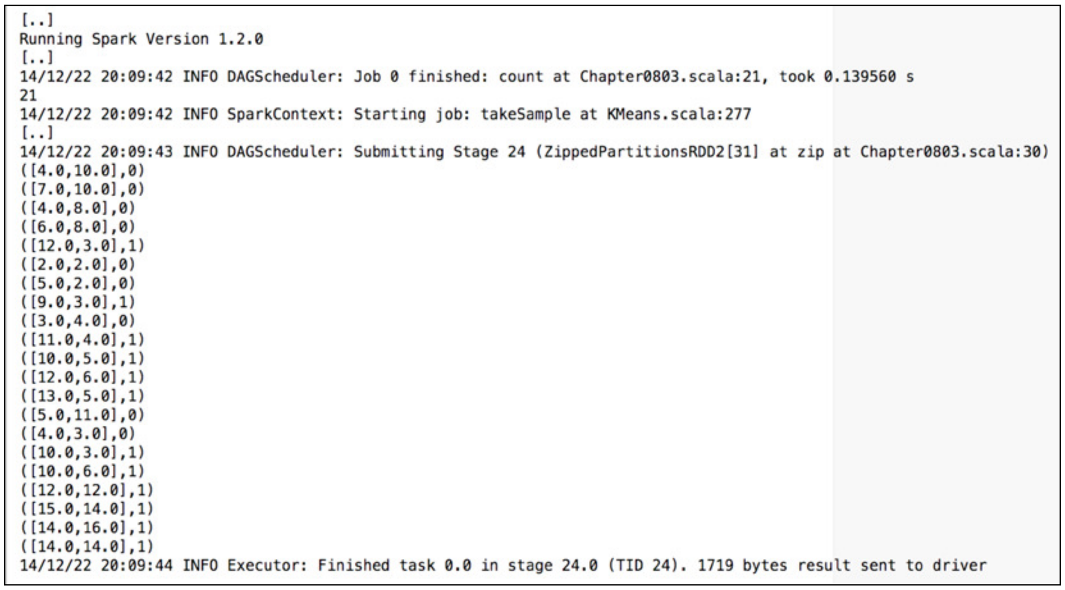
Let's use a sample data that has two dimensions *x* and *y*. The plot of the points would look like the following screenshot:



From the preceding graph, we can see that four clusters form one solution. Let's try with *k*=2 and *k*=4. Let's see how the Spark clustering algorithm handles this dataset and the groupings:

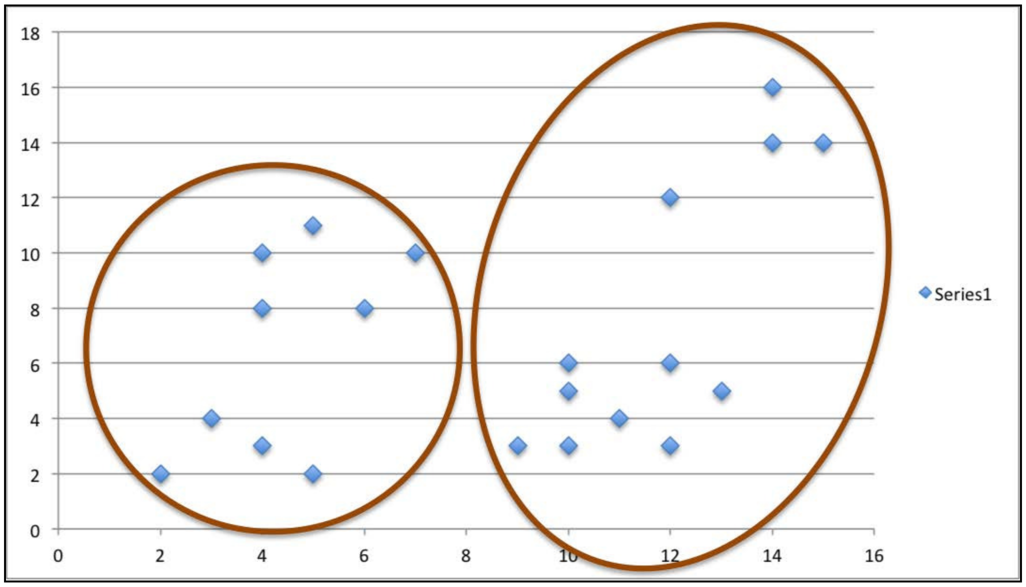
```
import org.apache.spark.SparkContext
import org.apache.spark.mllib.linalg.{Vector,Vectors}
import org.apache.spark.mllib.clustering.KMeans

object Chapter0803 {
  def parsePoints(inpLine : String) : Vector = {
    val values = inpLine.split(',')
    val x = values(0).toInt
    val y = values(1).toInt
    return Vectors.dense(x,y)
  }
  //

  def main(args: Array[String]): Unit = {
    val sc = new SparkContext("local","Chapter 8")
    println(s"Running Spark Version ${sc.version}")
    //
    val dataFile = sc.textFile("/Users/ksankar/bdtc-2014/cluster-
      points/cluster-points.csv")
    val points = dataFile.map(_.trim).filter( _.length > 1).
      map(line => parsePoints(line))
    //
    println(points.count())
    //
    var numClusters = 2
    val numIterations = 20
    var mdlKMeans = KMeans.train(points, numClusters,
      numIterations)
    //
    println(mdlKMeans.clusterCenters)
    //
    var clusterPred = points.map(x=>mdlKMeans.predict(x))
    var clusterMap = points.zip(clusterPred)
    //
    clusterMap.foreach(println)
    //
    clusterMap.saveAsTextFile("/Users/ksankar/bdtc-2014/cluster-
      points/2-cluster.csv")
    //
    // Now let us try 4 centers:
    //
    numClusters = 4
    mdlKMeans = KMeans.train(points, numClusters, numIterations)
    clusterPred = points.map(x=>mdlKMeans.predict(x))
    clusterMap = points.zip(clusterPred)
    clusterMap.saveAsTextFile("/Users/ksankar/bdtc-2014/cluster-
      points/4-cluster.csv")
    clusterMap.foreach(println)
  }
}
```

The results of the run would be as shown in the next screenshot (your run could give slightly different results):

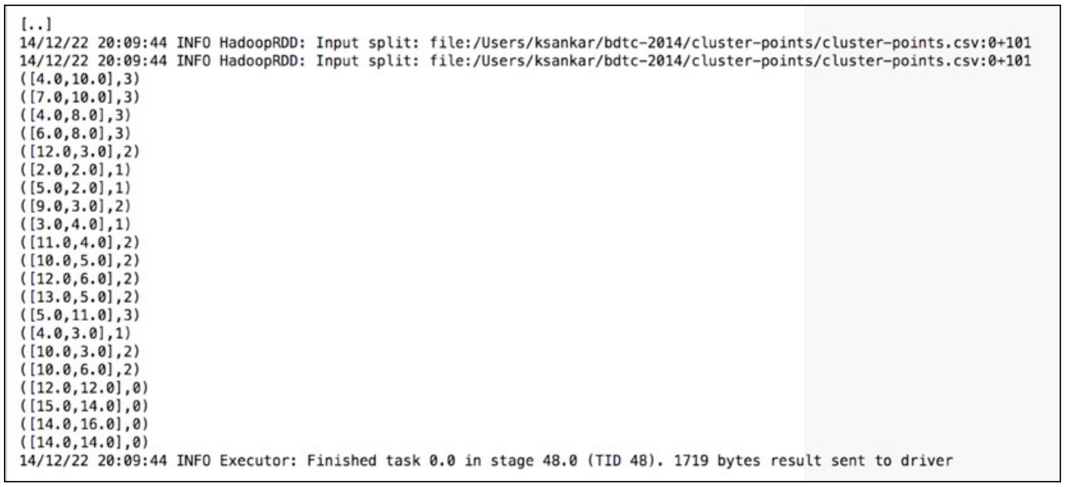


```
[..]
Running Spark Version 1.2.0
[..]
14/12/22 20:09:42 INFO DAGScheduler: Job 0 finished: count at Chapter0803.scala:21, took 0.139560 s
21
14/12/22 20:09:42 INFO SparkContext: Starting job: takeSample at KMeans.scala:277
[..]
14/12/22 20:09:43 INFO DAGScheduler: Submitting Stage 24 (ZippedPartitionsRDD2[31] at zip at Chapter0803.scala:30)
([4.0,10.0],0)
([7.0,10.0],0)
([4.0,8.0],0)
([6.0,8.0],0)
([12.0,3.0],1)
([2.0,2.0],0)
([5.0,2.0],0)
([9.0,3.0],1)
([3.0,4.0],0)
([11.0,4.0],1)
([10.0,5.0],1)
([12.0,6.0],1)
([13.0,5.0],1)
([5.0,11.0],0)
([4.0,3.0],0)
([10.0,3.0],1)
([10.0,6.0],1)
([12.0,12.0],1)
([15.0,14.0],1)
([14.0,16.0],1)
([14.0,14.0],1)
14/12/22 20:09:44 INFO Executor: Finished task 0.0 in stage 24.0 (TID 24). 1719 bytes result sent to driver
```

The *k*=2 graph shown in the next screenshot looks as expected:

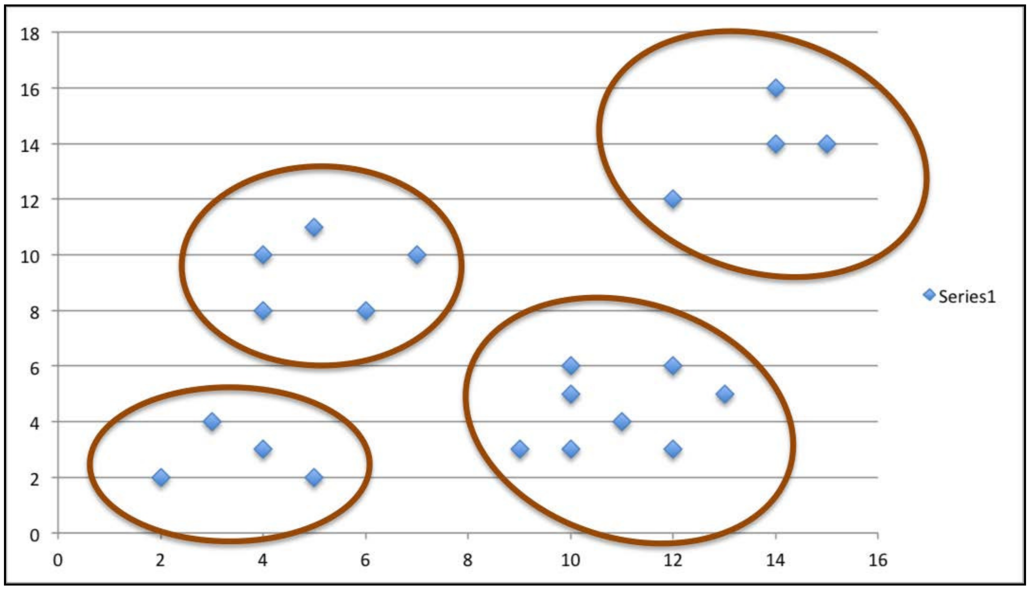


With *k=4* the results are as shown in the following screenshot:



```
[..]
14/12/22 20:09:44 INFO HadoopRDD: Input split: file:/Users/ksankar/bdtc-2014/cluster-points/cluster-points.csv:0+101
14/12/22 20:09:44 INFO HadoopRDD: Input split: file:/Users/ksankar/bdtc-2014/cluster-points/cluster-points.csv:0+101
([4.0,10.0],3)
([7.0,10.0],3)
([4.0,8.0],3)
([6.0,8.0],3)
([12.0,3.0],2)
([2.0,2.0],1)
([5.0,2.0],1)
([9.0,3.0],2)
([3.0,4.0],1)
([11.0,4.0],2)
([10.0,5.0],2)
([12.0,6.0],2)
([13.0,5.0],2)
([5.0,11.0],3)
([4.0,3.0],1)
([10.0,3.0],2)
([10.0,6.0],2)
([12.0,12.0],0)
([15.0,14.0],0)
([14.0,16.0],0)
([14.0,14.0],0)
14/12/22 20:09:44 INFO Executor: Finished task 0.0 in stage 48.0 (TID 48). 1719 bytes result sent to driver
```

The plot shown in the following screenshot confirms that the clusters are obtained as expected. Spark does understand clustering!



Bear in mind that the results could vary a little between runs because the clustering algorithm picks the centers randomly and grows from there. With *k=4*, the results are stable; but with *k=2*, there is room for partitioning the points in different ways. Try it out a few times and see the results.

Recommendation

The recommendation algorithms fall under five general mechanisms, namely, knowledge-based, demographic-based, content-based, collaborative filtering (item-based or user-based), and latent factor-based. Usually, the collaborative filtering is computationally intensive—Spark implements the **Alternating Least Square** (**ALS**) algorithm authored by Yehuda Koren, available at `http://dl.acm.org/citation.cfm?id=1608614`. It is user-based collaborative filtering using the method of learning latent factors, which can scale to a large dataset. Let's quickly use the `movielens` medium dataset to implement a recommendation using Spark.

There are some interesting RDD transformations. Apart from that, the code is not that complex, as shown next:

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._ // for implicit
  conversations
import org.apache.spark.mllib.recommendation.Rating
import org.apache.spark.mllib.recommendation.ALS

object Chapter0804 {

  def parseRating1(line : String) : (Int,Int,Double,Int) = {
    //println(x)
    val x = line.split("::")
    val userId = x(0).toInt
    val movieId = x(1).toInt
    val rating = x(2).toDouble
    val timeStamp = x(3).toInt/10
    return (userId,movieId,rating,timeStamp)
  }
  //
  def parseRating(x : (Int,Int,Double,Int)) : Rating = {
    val userId = x._1
    val movieId = x._2
    val rating = x._3
    val timeStamp = x._4 // ignore
    return new Rating(userId,movieId,rating)
  }
  //
```

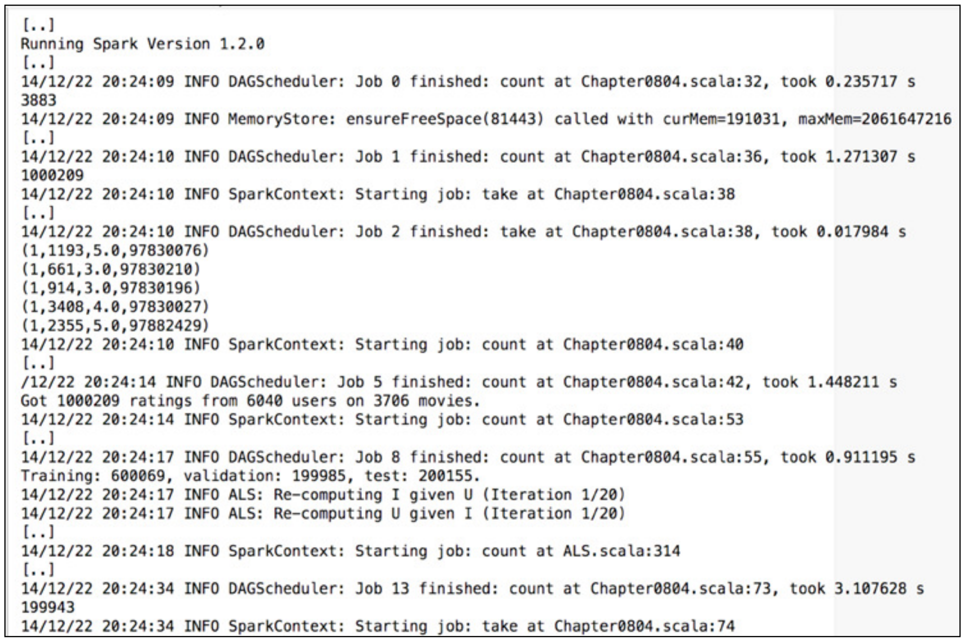
Now that we have the parsers in place, let's focus on the main program, as shown next:

```
def main(args: Array[String]): Unit = {
  val sc = new SparkContext("local","Chapter 8")
  println(s"Running Spark Version ${sc.version}")
  //
  val moviesFile = sc.textFile("/Users/ksankar/bdtc-
    2014/movielens/medium/movies.dat")
  val moviesRDD = moviesFile.map(line => line.split("::"))
  println(moviesRDD.count())
  //
  val ratingsFile = sc.textFile("/Users/ksankar/bdtc-
    2014/movielens/medium/ratings.dat")
  val ratingsRDD = ratingsFile.map(line => parseRating1(line))
  println(ratingsRDD.count())
  //
  ratingsRDD.take(5).foreach(println) // always check the RDD
  //
  val numRatings = ratingsRDD.count()
  val numUsers = ratingsRDD.map(r => r._1).distinct().count()
  val numMovies = ratingsRDD.map(r => r._2).distinct().count()
  println("Got %d ratings from %d users on %d movies.".
       format(numRatings, numUsers, numMovies))
```

Split the dataset into `training`, `validation`, and `test`. We can use any random dataset. But here we will use the last digit of the timestamp:

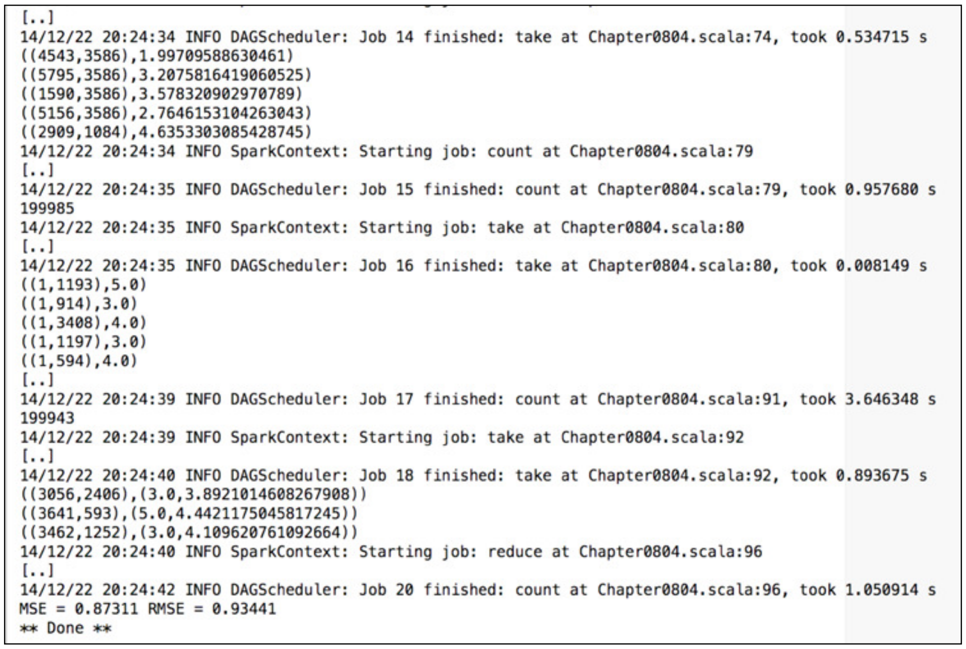
```
val trainSet = ratingsRDD.filter(x => (x._4 % 10) < 6)
  .map(x=>parseRating(x))
    val validationSet = ratingsRDD.filter(x => (x._4 % 10) >= 6 &
      (x._4 % 10) < 8).map(x=>parseRating(x))
    val testSet = ratingsRDD.filter(x => (x._4 % 10) >= 8)
      .map(x=>parseRating(x))
    println("Training: "+ "%d".format(trainSet.count()) +
      ", validation: " + "%d".format(validationSet.count()) + ",
        test: " + "%d".format(testSet.count()) + ".")
    //
    // Now train the model using the training set:
    val rank = 10
    val numIterations = 20
    val mdlALS = ALS.train(trainSet,rank,numIterations)
    //
    // prepare validation set for prediction
    //
    val userMovie = validationSet.map {
      case Rating(user, movie, rate) =>(user, movie)
    }
    //
    // Predict and convert to Key-Value PairRDD
    val predictions = mdlALS.predict(userMovie).map {
      case Rating(user, movie, rate) => ((user, movie), rate)
    }
    //
    println(predictions.count())
    predictions.take(5).foreach(println)
    //
    // Now convert the validation set to PairRDD:
    //
    val validationPairRDD = validationSet.map(r => ((r.user,
      r.product), r.rating))
    println(validationPairRDD.count())
    validationPairRDD.take(5).foreach(println)
    println(validationPairRDD.getClass())
    println(predictions.getClass())
    //
    // Now join the validation set with predictions.
    // Then we can figure out how good our recommendations are.
    // Tip:
    //   Need to import org.apache.spark.SparkContext._
    //   Then MappedRDD would be converted implicitly to PairRDD
    //
    val ratingsAndPreds = validationPairRDD.join(predictions)
    println(ratingsAndPreds.count())
    ratingsAndPreds.take(3).foreach(println)
    //
    val mse = ratingsAndPreds.map(r => {
      math.pow((r._2._1 - r._2._2),2)
    }).reduce(_+_) / ratingsAndPreds.count()
    val rmse = math.sqrt(mse)
    println("MSE = %2.5f".format(mse) + " RMSE = %2.5f"
      .format(rmse))
    println("** Done **")
  }
}
```

The run results, as shown in the next screenshot, are obtained as expected:



```
[..]
Running Spark Version 1.2.0
[..]
14/12/22 20:24:09 INFO DAGScheduler: Job 0 finished: count at Chapter0804.scala:32, took 0.235717 s
3883
14/12/22 20:24:09 INFO MemoryStore: ensureFreeSpace(81443) called with curMem=191031, maxMem=2061647216
[..]
14/12/22 20:24:10 INFO DAGScheduler: Job 1 finished: count at Chapter0804.scala:36, took 1.271307 s
1000209
14/12/22 20:24:10 INFO SparkContext: Starting job: take at Chapter0804.scala:38
[..]
14/12/22 20:24:10 INFO DAGScheduler: Job 2 finished: take at Chapter0804.scala:38, took 0.017984 s
(1,1193,5.0,97830076)
(1,661,3.0,97830210)
(1,914,3.0,97830196)
(1,3408,4.0,97830027)
(1,2355,5.0,97882429)
14/12/22 20:24:10 INFO SparkContext: Starting job: count at Chapter0804.scala:40
[..]
/12/22 20:24:14 INFO DAGScheduler: Job 5 finished: count at Chapter0804.scala:42, took 1.448211 s
Got 1000209 ratings from 6040 users on 3706 movies.
14/12/22 20:24:14 INFO SparkContext: Starting job: count at Chapter0804.scala:53
[..]
14/12/22 20:24:17 INFO DAGScheduler: Job 8 finished: count at Chapter0804.scala:55, took 0.911195 s
Training: 600069, validation: 199985, test: 200155.
14/12/22 20:24:17 INFO ALS: Re-computing I given U (Iteration 1/20)
14/12/22 20:24:17 INFO ALS: Re-computing U given I (Iteration 1/20)
[..]
14/12/22 20:24:18 INFO SparkContext: Starting job: count at ALS.scala:314
[..]
14/12/22 20:24:34 INFO DAGScheduler: Job 13 finished: count at Chapter0804.scala:73, took 3.107628 s
199943
14/12/22 20:24:34 INFO SparkContext: Starting job: take at Chapter0804.scala:74
```

Check the following screenshot as well:



```
[..]
14/12/22 20:24:34 INFO DAGScheduler: Job 14 finished: take at Chapter0804.scala:74, took 0.534715 s
((4543,3586),1.99709588630461)
((5795,3586),3.2075816419060525)
((1590,3586),3.578320902970789)
((5156,3586),2.7646153104263043)
((2909,1084),4.6353303085428745)
14/12/22 20:24:34 INFO SparkContext: Starting job: count at Chapter0804.scala:79
[..]
14/12/22 20:24:35 INFO DAGScheduler: Job 15 finished: count at Chapter0804.scala:79, took 0.957680 s
199985
14/12/22 20:24:35 INFO SparkContext: Starting job: take at Chapter0804.scala:80
[..]
14/12/22 20:24:35 INFO DAGScheduler: Job 16 finished: take at Chapter0804.scala:80, took 0.008149 s
((1,1193),5.0)
((1,914),3.0)
((1,3408),4.0)
((1,1197),3.0)
((1,594),4.0)
[..]
14/12/22 20:24:39 INFO DAGScheduler: Job 17 finished: count at Chapter0804.scala:91, took 3.646348 s
199943
14/12/22 20:24:39 INFO SparkContext: Starting job: take at Chapter0804.scala:92
[..]
14/12/22 20:24:40 INFO DAGScheduler: Job 18 finished: take at Chapter0804.scala:92, took 0.893675 s
((3056,2406),(3.0,3.8921014608267908))
((3641,593),(5.0,4.4421175045817245))
((3462,1252),(3.0,4.109620761092664))
14/12/22 20:24:40 INFO SparkContext: Starting job: reduce at Chapter0804.scala:96
[..]
14/12/22 20:24:42 INFO DAGScheduler: Job 20 finished: count at Chapter0804.scala:96, took 1.050914 s
MSE = 0.87311 RMSE = 0.93441
** Done **
```

Some more information is available at:

- The *Goodby MapReduce* article from Mahout News (`https://mahout.apache.org/`)
- `https://spark.apache.org/docs/latest/mllib-guide.html`
- A Collaborative Filtering ALS paper (`http://dl.acm.org/citation.cfm?id=1608614`)
- A good presentation on decision trees (`http://spark-summit.org/wp-content/uploads/2014/07/Scalable-Distributed-Decision-Trees-in-Spark-Made-Das-Sparks-Talwalkar.pdf`)
- A recommended hands-on exercise from Spark Summit 2014 (`https://databricks-training.s3.amazonaws.com/movie-recommendation-with-mllib.html`)

Summary

In this chapter, we looked at the most common machine learning algorithms. Naturally, ML is a vast subject and requires lot more study, experimentation, and practical experience on interesting data science problems. Two books that are relevant to Spark Machine Learning are Packt's own book *Machine Learning with Spark*, *Nick Pentreath*, and O'Reilly's *Advanced Analytics with Spark*, *Sandy Ryza*, *Uri Laserson*, *Sean Owen*, and *Josh Wills*. Both are excellent books that you can refer to.

10
Testing

Writing effective software without tests is quite challenging. Effective testing, especially in cases with slow end-to-end running times, such as distributed systems, can help improve developer effectiveness greatly. This chapter isn't going to try to convince you that you should be testing; however, if you really want to ride without a seat belt, that's fine too.

Testing in Java and Scala

For the sake of simplicity, this chapter covers using ScalaTest and JUnit as testing libraries. ScalaTest can be used to test both Scala and Java code and is the testing library currently used in Spark. To use ScalaTest with sbt, you need to add this to the `.sbt` file: `libraryDependencies += "org.scalatest" % "scalatest_2.10" % "2.0" % "test"`. JUnit is a popular testing framework for Java.

Making your code testable

If you have code that can be isolated from the RDD interaction or SparkContext interaction, that code can be tested using standard methodologies. While it can be quite convenient to use anonymous functions when writing Spark code, you cannot test them independently without the expensive overhead of setting up SparkContext. So the best practice is to write named functions. For example, in your CSV parser, you could take the following code:

- Scala code could be the following:

```
val splitLines = inFile.map(line => {
    val reader = new CSVReader(new StringReader(line))
    reader.readNext().map(_.toDouble)
  }
```

- Java code could be the following:

```
   JavaRDD<Integer[]> splitLines = inFile.flatMap(new
FlatMapFunction<String, Integer[]> (){
      public Iterable<Integer[]> call(String line) {
          ArrayList<Integer[]> result = new
            ArrayList<Integer[]>();
          try {
              CSVReader reader = new CSVReader(new
                 StringReader(line));
              String[] parsedLine = reader.readNext();
              Integer[] intLine = new
                Integer[parsedLine.length];
              for (int i = 0; i < parsedLine.length; i++) {
                intLine[i] = Integer.parseInt
                (parsedLine[i]);
              }
              result.add(intLine);
          } catch (Exception e) {
              errors.add(1);
          }
          return result;
       }
   }
  );
```

Instead of this, you could write the code as shown next:

```
 def parseLine(line: String): Array[Double] = {
      val reader = new CSVReader(new StringReader(line))
      reader.readNext().map(_.toDouble)
  }
```

Alternatively, in Java, you could write the code as shown here:

```
public class JavaLoadCsvTestable {
    public static class ParseLine extends Function<String, Integer[]>
{
    public Integer[] call(String line) throws Exception {
        CSVReader reader = new CSVReader(new StringReader(line));
        String[] parsedLine = reader.readNext();
        Integer[] intLine = new Integer[parsedLine.length];
        for (int i = 0; i < parsedLine.length; i++) {
          intLine[i] = Integer.parseInt(parsedLine[i]);
        }
        return intLine;
    }
  }
}
```

You can then test it without having to worry about any Spark specific setup or logic as shown in the following code:

```
import org.scalatest.FunSuite
import org.scalatest.matchers.ShouldMatchers

class TestableLoadCsvExampleSuite extends FunSuite with
  ShouldMatchers {
    test("should parse a csv line with numbers") {
      TestableLoadCsvExample.parseLine("1,2") should equal
        (Array[Double](1.0,2.0))
      TestableLoadCsvExample.parseLine("100,-1,1,2,2.5") should
        equal (Array[Double](100,-1,1.0,2.0,2.5))
    }
    test("should error if there is a non-number") {
      evaluating { TestableLoadCsvExample.parseLine("pandas")  }
        should produce [NumberFormatException]
    }
}
```

Alternatively, to test the Java code, you would do something like the following code (note that the test is still written in Scala; don't worry as we will look at JUnit tests later):

```
class JavaLoadCsvExampleSuite extends FunSuite with ShouldMatchers {

    test("should parse a csv line with numbers") {
      val parseLine = new JavaLoadCsvTestable.ParseLine();
      parseLine.call("1,2") should equal (Array[Integer](1,2))
      parseLine.call("100,-1,1,2,2") should equal (Array[Integer]
        (100,-1,1,2,2))
    }
    test("should error if there is a non-integer") {
      val parseLine = new JavaLoadCsvTestable.ParseLine();
      evaluating { parseLine.call("pandas")  } should produce
        [NumberFormatException]
      evaluating {parseLine.call("100,-1,1,2.2,2") should equal
        (Array[Integer](100,-1,1,2,2)) } should produce
        [NumberFormatException]
    }
}
```

Testing interactions with SparkContext

You may, however, remember that you later extended the CSV parser to increment counters on invalid input to gracefully handle failures. To verify that behavior, you could provide mock counters and other mock objects for the Spark components you use. You are restricted to only test the parts of the code that do not depend on Spark. Instead, you could re-factor the code to make the core testable without Spark and to do a more complete test using a provided SparkContext, as shown:

```
object MoreTestableLoadCsvExample {
  def parseLine(line: String): Array[Double] = {
    val reader = new CSVReader(new StringReader(line))
    reader.readNext().map(_.toDouble)
  }
  def handleInput(invalidLineCounter: Accumulator[Int], inFile:
    RDD[String]): RDD[Double] = {
    val numericData = inFile.flatMap(line => {
      try {
    Some(parseLine(line))
      } catch {
    case _ => {
      invalidLineCounter += 1
      None
    }
      }
    })
    numericData.map(row => row.sum)
  }

  def main(args: Array[String]) {
    if (args.length != 2) {
      System.err.println("Usage: TestableLoadCsvExample <master>
        <inputfile>")
      System.exit(1)
    }
    val master = args(0)
    val inputFile = args(1)
    val sc = new SparkContext(master, "Load CSV Example",
                      System.getenv("SPARK_HOME"),
                      Seq(System.getenv("JARS")))
    sc.addFile(inputFile)
    val inFile = sc.textFile(inputFile)
    val invalidLineCounter = sc.accumulator(0)
    val summedData = handleInput(invalidLineCounter, inFile)
    println(summedData.collect().mkString(","))
```

```
      println("Errors: "+invalidLineCounter)
      println(summedData.stats())
    }

  }
```

This does have the downside of requiring that your tests run serially, else sbt (or other build infrastructure) may try to launch multiple Spark contexts at the same time, which will cause confusing error messages. We can force the tests to execute sequentially in sbt with `parallelExecution in Test := false`.

We test this by using the following code:

```
import org.apache.spark._
import org.apache.spark.SparkContext._
import org.scalatest.FunSuite
import org.scalatest.matchers.ShouldMatchers

class MoreTestableLoadCsvExampleSuite extends FunSuite with
  ShouldMatchers {
  test("summ data on input") {
    val sc = new SparkContext("local", "Load CSV Example")
    val counter = sc.accumulator(0)
    val input = sc.parallelize(List("1,2","1,3"))
    val result = MoreTestableLoadCsvExample.handleInput(counter,
      input)
    result.collect() should equal (Array[Int](3,4))
  }
  test("should parse a csv line with numbers") {
    MoreTestableLoadCsvExample.parseLine("1,2") should equal
      (Array[Double](1.0,2.0))
    MoreTestableLoadCsvExample.parseLine("100,-1,1,2,2.5") should
      equal (Array[Double](100,-1,1.0,2.0,2.5))
  }
  test("should error if there is a non-number") {
    evaluating { MoreTestableLoadCsvExample.parseLine("pandas")  }
      should produce [NumberFormatException]
  }
}
```

In Java, you can test with the following code:

```
public class JavaLoadCsvMoreTestable {
    public static class ParseLineWithAcc extends
      FlatMapFunction<String, Integer[]> {
    Accumulator<Integer> acc;
    ParseLineWithAcc(Accumulator<Integer> acc) {
        this.acc = acc;
    }
   public Iterable<Integer[]> call(String line) throws Exception {
        ArrayList<Integer[]> result = new ArrayList<Integer[]>();
        try {
            CSVReader reader = new CSVReader(new
              StringReader(line));
        String[] parsedLine = reader.readNext();
        Integer[] intLine = new Integer[parsedLine.length];
        for (int i = 0; i < parsedLine.length; i++) {
            intLine[i] = Integer.parseInt(parsedLine[i]);
        }
        result.add(intLine);
        } catch (Exception e) {
        acc.add(1);
        }
        return result;
    }
    }
    public static JavaDoubleRDD processData(Accumulator<Integer>
      acc, JavaRDD<String> input) {
    JavaRDD<Integer[]> splitLines = input.flatMap(new
      ParseLineWithAcc(acc));
    JavaDoubleRDD summedData = splitLines.map(new
      DoubleFunction<Integer[]>() {
         public Double call(Integer[] in) {
            Double ret = 0.;
            for (int i = 0; i < in.length; i++) {
                ret += in[i];
            }
            return ret;
        }
      }
    );
return summedData;
    }
```

You can test this in Scala code as shown here (note that we add an invalid input for the counter here):

```
class JavaLoadCsvMoreTestableSuite extends FunSuite with
  ShouldMatchers {
  test("sum data on input") {
    val sc = new JavaSparkContext("local", "Load Java CSV test")
    val counter: Accumulator[Integer] = sc.intAccumulator(0)
    val input: JavaRDD[String] =
      sc.parallelize(List("1,2","1,3","murh"))
    val javaLoadCsvMoreTestable = new JavaLoadCsvMoreTestable();
    val resultRDD = JavaLoadCsvMoreTestable.
      processData(counter,input)
    resultRDD.cache();
    val resultCount = resultRDD.count()
    val result = resultRDD.collect().toArray()
    resultCount should equal (2)
    result should equal (Array[Double](3.0, 4.0))
    counter.value should equal (1)
    sc.stop()
  }
}
```

You can test this in Java with Junit4, as shown in the following code:

```
package pandaspark.examples;

import org.apache.spark.*;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaDoubleRDD;
import org.scalatest.FunSuite;
import org.scalatest.matchers.ShouldMatchers;

import static org.junit.Assert.assertEquals;
import org.junit.Test;
import org.junit.Ignore;
import org.junit.runner.RunWith;
import org.junit.runners.JUnit4;

import java.util.Arrays;
import java.util.List;
import java.util.ArrayList;


@RunWith(JUnit4.class)
```

```
public class JavaLoadCsvMoreTestableSuiteJunit {
    @Test
    public void testSumDataOnInput() {
    JavaSparkContext sc = new JavaSparkContext("local", "Load Java
      CSV test");
    Accumulator<Integer> counter = sc.intAccumulator(0);
    String[] inputArray = {"1,2","1,3","murh"};
    JavaRDD<String> input = sc.parallelize
      (Arrays.asList(inputArray));
    JavaDoubleRDD resultRDD = JavaLoadCsvMoreTestable.
      processData(counter, input);
    long resultCount = resultRDD.count();
    assertEquals(resultCount, 2);
    int errors = counter.value();
    assertEquals(errors, 1);
    sc.stop();
  }
}
```

Testing in Python

Python testing of Spark is very similar in concept to testing in Java and Scala, but the testing libraries are a bit different. PySpark uses both doctest and unittest to test itself. doctest makes it easy to create tests based on the expected output of code run in the Python interpreter. We can run the tests using the following commands:

```
export SPARK_TESTING=1
export PYSPARK_DOC_TEST=1
bin/pyspark [pathtocode]
```

By taking the `wordcount.py` example from Spark and factoring out `countWords`, you can test the word count functionality using doctest. Some doctest examples are shown next:

```
"""
>>> from pyspark.context import SparkContext
>>> sc = SparkContext('local', 'test')
>>> b = sc.parallelize(["pandas are awesome","and ninjas are also
awesome"])
>>> countWords(b)
[('also', 1), ('and', 1), ('are', 2), ('awesome', 2), ('ninjas', 1),
('pandas', 1)]
"""
```

```
import sys
from operator import add

from pyspark import SparkContext

def countWords(lines):
    counts = lines.flatMap(lambda x: x.split(' ')) \
                  .map(lambda x: (x, 1)) \
                  .reduceByKey(add)
    return sorted(counts.collect())


if __name__ == "__main__":
    if len(sys.argv) < 3:
        print >> sys.stderr, \
            "Usage: PythonWordCount <master> <file>"
        exit(-1)
    sc = SparkContext(sys.argv[1], "PythonWordCount")
    lines = sc.textFile(sys.argv[2], 1)
    output = countWords(lines)
    for (word, count) in output:
        print "%s : %i" % (word, count)
```

Note about doctest

You put the test in between triple quotes. The testing code is prefixed with >>> as if it's running in the Python shell. The expected output that would be seen is added exactly as if it's returned in the Python shell.

We can also test something similar to our Java and Scala programs, as shown next:

```
"""
>>> from pyspark.context import SparkContext
>>> sc = SparkContext('local', 'test')
>>> b = sc.parallelize(["1,2","1,3"])
>>> handleInput(b)
[3, 4]
"""

import sys
from operator import add

from pyspark import SparkContext
def handleInput(lines):
    data = lines.map(lambda x: sum(map(int, x.split(','))))
    return sorted(data.collect())


if __name__ == "__main__":
    if len(sys.argv) < 3:
        print >> sys.stderr, \
            "Usage: PythonLoadCsv <master> <file>"
        exit(-1)
    sc = SparkContext(sys.argv[1], "PythonLoadCsv")
    lines = sc.textFile(sys.argv[2], 1)
    output = handleInput(lines)
    for sum in output:
        print sum
```

Some more information can be found at the following sites:

- `http://blog.quantifind.com/posts/spark-unit-test/`
- `http://www.scalatest.org/`
- `http://junit.org/`
- `http://docs.python.org/2/library/unittest.html`
- `http://docs.python.org/2/library/doctest.html`

Summary

This chapter discussed how to structure your code so that it is testable as well as the testing framework that is used within Spark. Effective testing can save large amounts of debugging time, which can be especially painful in large distributed systems. In the next chapter, we will look at some tips and tricks such as tuning and securing Spark.

11
Tips and Tricks

As discussed in the earlier chapters, you have the tools to build and test Spark jobs as well as set up a Spark cluster to run them on, so now it's time to figure out how to make the most of your time as a Spark developer. The Spark documentation includes good tips on tuning and is available at `http://spark.apache.org/docs/latest/tuning.html`.

Where to find logs

Spark has very useful logs to figure out what's going on when things are not going as expected. Spark keeps a per machine log on each machine by default in the `SPARK_HOME/work` subdirectory. Spark's web UI provides a convenient place to see `STDOUT` and `STDERR` of each job, running and completed jobs, separated out per worker.

Concurrency limitations

Spark's concurrency for operations is limited by the number of partitions. Conversely, having too many partitions can cause excess overhead by launching too many tasks. If you have too many partitions, you can shrink it by using the `coalesce(numPartitions,shuffle)` method. The `coalesce` method is a good method to pack and rebalance your RDDs (for example, after a filter operation where you have less data after the action). If the new number of partitions is more than what you have now, set `shuffle=True`, else set `shuffle=false`. While creating a new RDD, you can specify the number of partitions to be used. Also, the grouping/joining mechanism on RDDs of pairs can take the number of partitions or a custom `partitioner` class. The default number of partitions for new RDDs is controlled by `spark.default.parallelism`, which also controls the number of tasks used by `groupByKey` and other shuffle operations that need shuffling.

Memory usage and garbage collection

To measure the impact of garbage collection, you can ask the JVM to print details about the garbage collection. You can do this by adding `-verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps` to your `SPARK_JAVA_OPTS` in `conf/spark-env.sh`. You can also include the `-Xloggc` option to print the log messages to a separate file so that log messages are kept separate. The details will then be printed to the standard out when you run your job, which will be available as described in the first section of this chapter.

If you find that your Spark cluster uses too much time collecting garbage, you can reduce the amount of space used for RDD caching by changing `spark.storage.memoryFraction`; here, the default is `0.6`. If you are planning to run Spark for a long time on a cluster, you may wish to enable `spark.cleaner.ttl`. By default, Spark does not clean up any metadata (stages generated, tasks generated, and so on); set this to a non-zero value in seconds to clean up the metadata after that length of time. The documentation page (`https://spark.apache.org/docs/latest/configuration.html`) has the default settings and details about all the configuration options.

You can also control the RDD storage level if you find that you use too much memory. I usually use `top` to see the memory consumption of the processes. If your RDDs don't fit within memory and you still wish to cache them, you can try using a different storage level shown as follows (also check the documentation page for the latest information on RDD persistence options at `http://spark.apache.org/docs/latest/programming-guide.html#rdd-persistence`):

- `MEMORY_ONLY`: This stores the entire RDD in memory if it can, which is the default
- `MEMORY_AND_DISK`: This stores each partition in memory if it can fit; else it stores it on disk
- `DISK_ONLY`: This stores each partition on disk regardless of whether it can fit in memory

These options are set when you call the persist function (`rdd.persist()`) on your RDD. By default, the RDDs are stored in a deserialized form, which requires less parsing. We can save space by adding `_SER` to the storage level (for example, `MEMORY_ONLY_SER`, `MEMORY_AND_DISK_SER`), in which case Spark will serialize the data to be stored, which normally saves some space but increases the execution time.

Serialization

Spark supports different serialization mechanisms; the choice is a trade-off between speed, space efficiency, and full support of all Java objects. If you are using the serializer to cache your RDDs, you should strongly consider a fast serializer. The default serializer uses Java's default serialization. The KyroSerializer is much faster and generally uses about one tenth of the memory as the default serializer. You can switch the serializer by setting `spark.serializer` to `spark.KryoSerializer`. If you want to use KyroSerializer, you need to make sure that the classes are serializable by KyroSerializer. Spark provides a trait `KryoRegistrator`, which you can extend to register your classes with Kyro, as shown in the following code:

```
class Reigstrer extends spark.KyroRegistrator {
    override def registerClasses(kyro: Kyro) {
              kyro.register(classOf[MyClass])
    }
}
```

Take a look at `https://code.google.com/p/kryo/#Quickstart` to figure out how to write custom serializers for your classes if you need something customized. You can substantially decrease the amount of space used for your objects by customizing your serializers. For example, rather than writing out the full class name, you can give them an integer ID by calling `kyro.register(classOf[MyClass],100)`.

IDE integration

For Emacs users, the ENSIME sbt plugin is a good addition. **ENhanced Scala Interaction Mode for Emacs** (**ENSIME**) provides many features that are available in IDEs such as error checking and symbol inspection. You can install the latest ENSIME from `https://github.com/aemoncannon/ensime/downloads` (make sure you choose the one that matches your Scala version). Or, you can run the following commands:

```
wget https://github.com/downloads/aemoncannon/ensime/ ensime_2.10.0-RC3-0.9.8.2.tar.gz


tar -xvf ensime_2.10.0-RC3-0.9.8.2.tar.gz
```

In your `.emacs`, add this:

```
;; Load the ensime lisp code...
(add-to-list 'load-path "ENSIME_ROOT/elisp/")
(require 'ensime)
```

```
;; This step causes the ensime-mode to be started whenever
;; scala-mode is started for a buffer. You may have to customize ;;
this step if you're not using the standard scala mode.
(add-hook 'scala-mode-hook 'ensime-scala-mode-hook)
```

You can then add the ENSIME sbt plugin to your project (in `project/plugins.sbt`):

```
addSbtPlugin("org.ensime" % "ensime-sbt-cmd" % "0.1.0")
```

You should then run the following commands:

```
sbt
> ensime generate
```

If you are using Git, you will probably want to add `.ensime` to the `.gitignore` file if it isn't already present.

If you have an IntelliJ, a similar plugin exists called sbt-idea, which can be used to generate IntelliJ idea files. You can add the IntelliJ sbt plugin to your project (in `project/plugins.sbt`) like this:

```
addSbtPlugin("com.github.mpeltonen" % "sbt-idea" % "1.5.1")
```

You should then run the following commands:

```
sbt
> gen-idea
```

This will generate the idea file, which can be loaded into IntelliJ.

Eclipse users can also use sbt to generate Eclipse project files with the sbteclipse plugin. You can add the Eclipse sbt plugin to your project (in `project/plugins.sbt`) like this:

```
addSbtPlugin("com.typesafe.sbteclipse" % "sbteclipse-plugin" %
  "2.3.0")
```

You should then run the following commands:

```
sbt
> eclipse
```

This will generate the Eclipse project files and you can then import them into your Eclipse project using the Import Wizard in Eclipse. Eclipse users might also find the spark-plug project useful, which can be used to launch clusters from within Eclipse.

An import step is to add `spark-assembly-1.2.0-hadoop2.6.0.jar` in your Java build path or Maven dependency. Pay attention so you match the Spark version number (1.2.0) with the Hadoop version number (2.6.0).

Using Spark with other languages

If you find yourself wanting to work with your RDD in another language, there are a few options available for you. From Java/Scala you can try using JNI, and with Python you can use the FFI. Sometimes however, you will want to work with a language that isn't C or work with an already compiled program. In that case, the easiest thing to do is to use the pipe interface that is available in all three of the APIs. The stream API works by taking the RDD and serializing it to strings and then piping it to the specified program. If your data happens to be plain strings, this is very convenient, but if it's not so, you will need to serialize your data in such a way that it can be understood on either side. JSON or protocol buffers can be good options for this depending on how structured your data is.

A quick note on security

Another important consideration in your Spark setup is security. If you are using Spark on EC2 with the default scripts, you will notice that the access to your Spark cluster is restricted. This is a good idea to do even if you aren't running inside of EC2 since your Spark cluster will likely have access to the data you would rather not share with the world (and even if it doesn't have it, you probably don't want to allow arbitrary code execution by strangers). If your Spark cluster is already on a private network, that is great, otherwise you should talk with your system administrator about setting up some IPtables rules to restrict access.

Community developed packages

A new package index site (`http://spark-packages.org/`) has a lot of packages and libraries that work with Apache Spark. It's an essential site to visit and make use of.

Mailing lists

Probably the most useful tip to finish this chapter with is that the Spark user's mailing list is an excellent source of up-to-date information about other people's experiences in using Spark. The best place to get information on meetups, slides, and so forth is `https://spark.apache.org/community.html`. The two Spark users mailing lists are `user@spark.apache.org` and `dev@spark.apache.org`.

Some more information can be found at the following sites:

- `http://blog.quantifind.com/posts/logging-post/`
- `http://jawher.net/2011/01/17/scala-development-environment-emacs-sbt-ensime/`
- `https://www.assembla.com/spaces/liftweb/wiki/Emacs-ENSIME`
- `https://github.com/shivaram/spark-ec2/blob/master/ganglia/init.sh`
- `https://spark.apache.org/docs/latest/tuning.html`
- `http://spark.apache.org/docs/latest/running-on-mesos.html`
- `http://kryo.googlecode.com/svn/api/v2/index.html`
- `https://code.google.com/p/kryo/`
- `http://scala-ide.org/download/current.html`
- `http://syndeticlogic.net/?p=311`
- `http://mail-archives.apache.org/mod_mbox/incubator-spark-user/`
- `https://groups.google.com/forum/?fromgroups#!forum/spark-users`

Summary

That wraps up some common things that you can use to help improve your Spark development experience. I wish you the best of luck with your Spark projects; now go and solve some fun problems! :)

Index