



# Spark

**Succinctly**

by Marko Švaljek

# Spark Succinctly

---

By  
Marko Švaljek

Foreword by Daniel Jebaraj



Copyright © 2015 by Syncfusion, Inc.  
2501 Aerial Center Parkway  
Suite 200  
Morrisville, NC 27560  
USA  
All rights reserved.

## **Important licensing information. Please read.**

This book is available for free download from [www.syncfusion.com](http://www.syncfusion.com) on completion of a registration form.

If you obtained this book from any other source, please register and download a free copy from [www.syncfusion.com](http://www.syncfusion.com).

This book is licensed for reading only if obtained from [www.syncfusion.com](http://www.syncfusion.com).

This book is licensed strictly for personal or educational use.

Redistribution in any form is prohibited.

The authors and copyright holders provide absolutely no warranty for any information provided.

The authors and copyright holders shall not be liable for any claim, damages, or any other liability arising from, out of, or in connection with the information in this book.

Please do not use this book if the listed terms are unacceptable.

Use shall constitute acceptance of the terms listed.

SYNCFUSION, SUCCINCTLY, DELIVER INNOVATION WITH EASE, ESSENTIAL, and .NET ESSENTIALS are the registered trademarks of Syncfusion, Inc.

**Technical Reviewer:** James McCaffrey

**Copy Editor:** Darren West, content producer, Syncfusion, Inc.

**Acquisitions Coordinator:** Hillary Bowling, marketing coordinator, Syncfusion, Inc.

**Proofreader:** Tres Watkins, content development manager, Syncfusion, Inc.

# Table of Contents

<b>The Story behind the <i>Succinctly</i> Series of Books .....</b>	<b>6</b>
<b>About the Author.....</b>	<b>8</b>
<b>Introduction .....</b>	<b>9</b>
Basics of Big Data Processing.....	10
A Brief History of Spark.....	15
Spark Overview.....	16
<b>Chapter 1 Installing Spark.....</b>	<b>18</b>
Installing Spark Prerequisites on Linux.....	18
Installing Java .....	18
Installing Python .....	19
Installing Scala.....	20
Installing Spark Prerequisites on Windows.....	21
Installing Java .....	21
Installing Python .....	24
Installing Scala.....	27
Spark Download and Configuration .....	32
<b>Chapter 2 Hello Spark.....</b>	<b>35</b>
Counting Text Lines Containing Text.....	35
Scala Example .....	36
Python Example.....	37
Java Example and Development Environment Setup .....	38
Counting Word Occurrences .....	51
<b>Chapter 3 Spark Internals.....</b>	<b>56</b>
Resilient Distributed Dataset (RDD) .....	56

Caching RDDs .....	62
Pair RDDs .....	66
Aggregating Pair RDDs .....	69
Data Grouping .....	74
Sorting Pair RDDs .....	78
Join, Intersect, Union and Difference Operations on Pair RDDs.....	81
<b>Chapter 4 Data Input and Output with Spark .....</b>	<b>92</b>
Working with Text Files.....	92
Submitting Scala and Python Tasks to Spark.....	96
Working with JSON files .....	98
Spark and Cassandra .....	105
<b>Conclusion.....</b>	<b>111</b>

# The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President  
Syncfusion, Inc.

**S**taying on the cutting edge  
As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

## Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

## The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

## The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

## Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

## Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

## Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at [succinctly-series@syncfusion.com](mailto:succinctly-series@syncfusion.com).

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!



# About the Author

Marko Švaljek works as a software developer, and in his ten years of experience he has worked for the leading financial and telecom companies in southeast Europe with emphasis on the Internet of Things, mobile banking, and e-commerce solutions. The main focus of his interest in the past couple of years has been the Internet of Things. Until now, Marko had only authored two books, *Cassandra Succinctly* and *Arduino Succinctly*. In the context of the Internet of Things, the first book deals with how to store persistent data generated by various devices, and the second one focuses on how to create the sensors that actually generate various readings in the first place.

Apart from generating and storing the data, one of the most interesting aspects of the Internet of Things Marko is so drawn to is analyzing the data that comes in. For quite some time now, Marko has used Apache Spark as his favorite big data processing framework, and this book is the third part in Marko's Internet of Things saga.



# Introduction

Many of the leading companies in the world today face the problem of big data. There are various definitions by various authors on what big data actually is. To be honest, it's a vague definition, and most of the authors in the field have their own. The scope of this book is too limited to go into discussions and explanations, but if you are a seasoned developer in the field, you probably have your own view of what big data is.

If you are a newcomer to the field, perhaps the easiest way to explain the concept is that it's the data that can't be handled with traditional computing technologies, which mostly used single machines to process data. I won't go into corner cases like, "what if you had a really powerful computer" and so on. The easiest way to think about big data is that it's data that can't be processed or stored by a single machine. When it comes to the whole big data use case, the flow usually looks like the following figure:

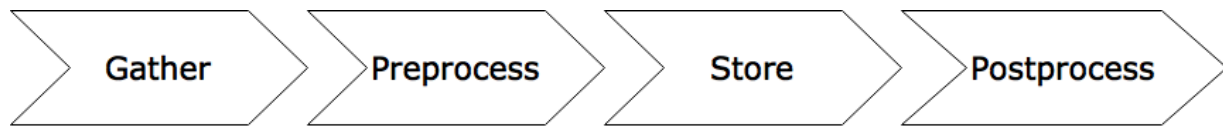


Figure 1: Big Data Information Flow

To create big data, first you have to gather the data. The gathered data can come from anything, i.e. weather stations, social media, or even another computer process. If you have ever worked with large data sets, then you might know that the data that we ingest is often not in the format that we expect it to be or might contain various invalid values. Data is usually preprocessed so that we can maintain the usability of data in our system. The next step in the information handling flow is storing the data. We'll explain the store step in the chapters to come. For now, it's sufficient to say that the store step is simply holding data that is going to be processed at a later point in time. The final step is usually post-processing. This book will be mostly about the post-processing step because that's the way Spark is most often used. Now, don't get me wrong: you can use Spark in any of the steps. But most of the time, Spark is used as the last step in the big data information flow. This is a bit of a simplified description of big data in the context of Spark usage, but it's enough to get you going for the chapters to come. Let's have a look at the high-level overview of Spark usage:

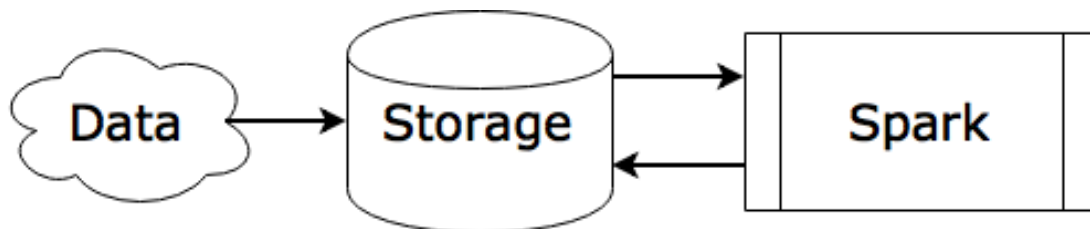


Figure 2: Spark in the Big Data Flow

Note that on the figure Spark takes input data and produces output data using the same storage. In practice this can vary, as data can come in from multiple data sources. Most of the examples provided in this book will be relatively simple when it comes to data output and the number of data processing results because the focus is on how to gather and process the data. Data output is a relatively easier step, but do remember that in practice Spark makes data output to other storage systems, especially in the context of big data production systems. This doesn't mean that you can't have any fun with Spark on your own or that you can only use it on multiple machines with multiple cores. Spark can also be used to make various data explorations on your own by using the shells bundled with Spark.

There are many programming languages, and Spark as a technology is not bound to just one of them. The languages supported out of the box in the Spark 1.4 release are:

- Scala – multi-paradigm programming language for Java Virtual Machine
- Java – very popular class-based, object-oriented programming language
- Python – general purpose programming language, very popular among academics
- R – programming language for statistical computing

If you want to use Spark with other programming technologies, there is the concept of pipes, with which Spark can read and write data from standard Unix streams. We'll discuss this in later chapters of the book; this is a very high-level overview of what Spark is and how it can be used in the context of big data processing. Let's dive into the basics of big data processing before getting to know more about Spark.

## Basics of Big Data Processing

In the previous section, we mentioned that the simplest definition of big data would be that it is data that can't be processed on a single computer, at least not within a reasonable timeframe. Theoretically, one could solve the problem by employing a faster computer with more memory, but that kind of specialized machine costs a great deal of money. The next logical step would be to split the computing among multiple commodity computers. But splitting the data and the processing among them is not as simple as it might seem:

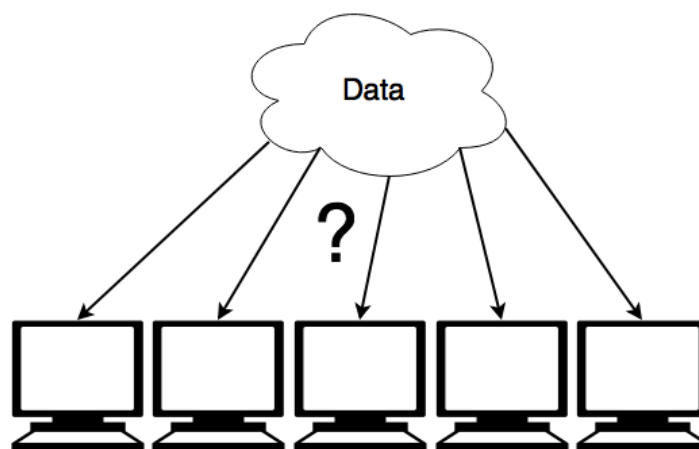
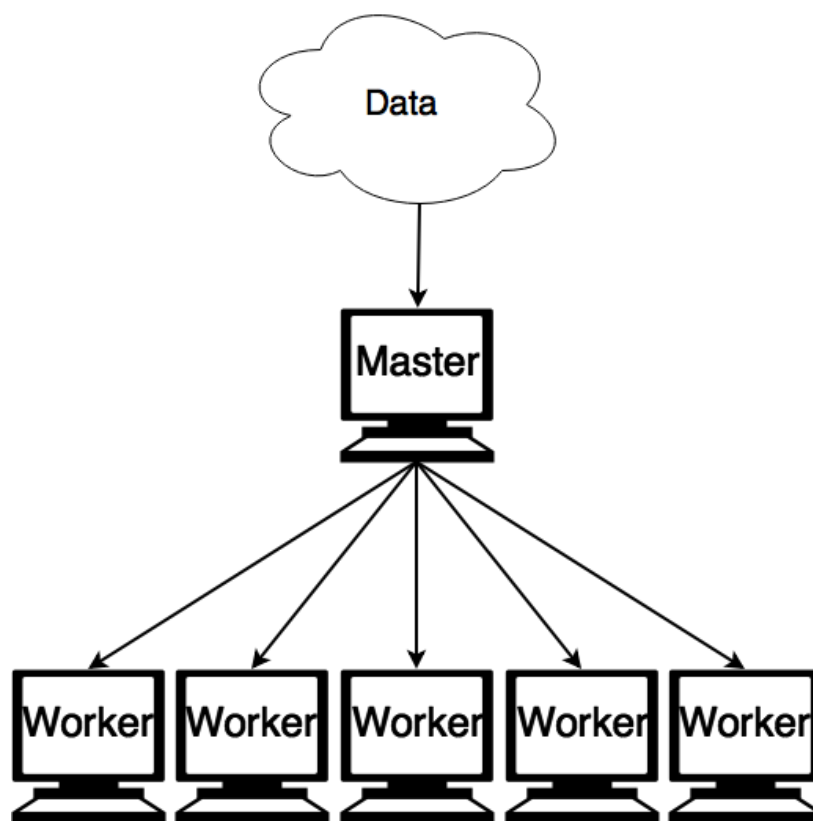


Figure 3: The Problem of Splitting Data Processing

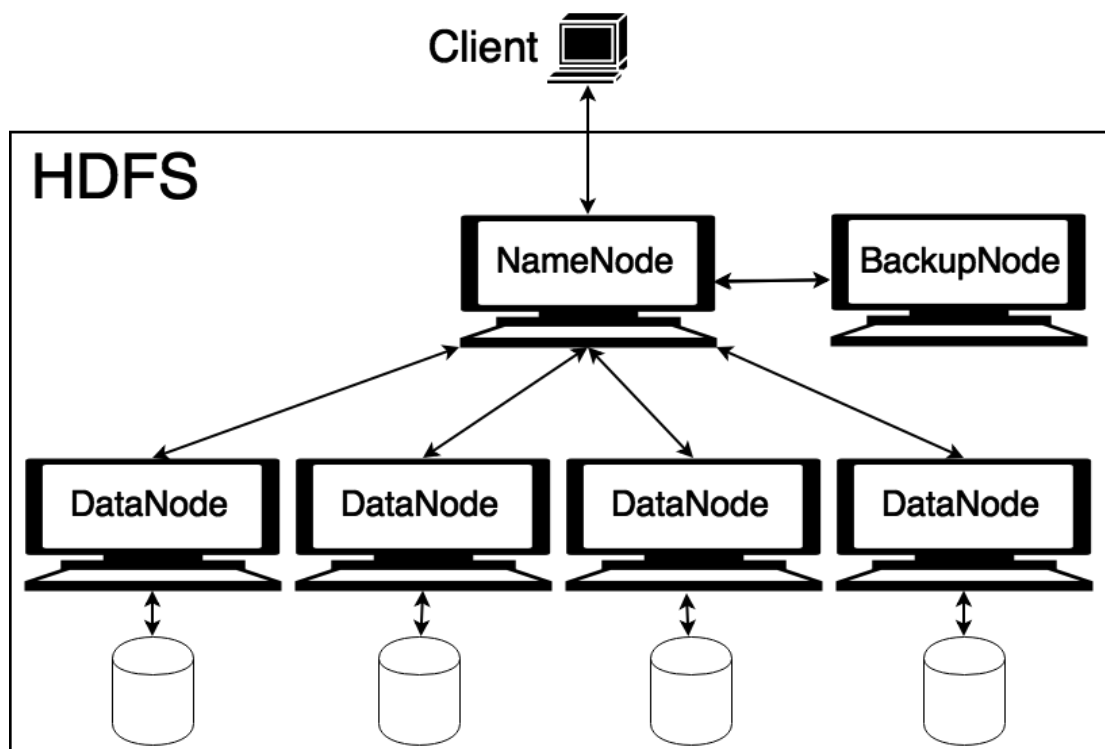
There's a great amount of scheduling related to processing time and the I/O operations to do such a thing. Letting nodes negotiate themselves to split the tasks among them would imply a great deal of complex and hard-to-debug technologies with significant overhead and negotiation time among the nodes. To avoid all those problems, most modern systems use master-worker architecture. The master node determines what data is processed on what worker node and when, as show on the following figure:



*Figure 4: Master-Worker Architecture*

Some of you might think that there is some sort of a bottleneck when using master-worker architecture, and that is true in some solutions but most of the modern ones have the master node in charge only for delegating the work and monitoring the results. The worker nodes fetch the data themselves from the source according to the instructions from the master. Up until now the data was simply shown as a small cloud and we didn't go into it. Earlier we mentioned that big data also refers to the data that can't actually fit on a single computer. That's why data is often saved on a distributed file system. In the world of big data, the most popular distributed file system at the moment is Hadoop Distributed File System (HDFS). Some of you might now think something like: wait, this book is supposed to be about Spark and not Hadoop. And some of you might not even know what Hadoop is. If you want to find out more about Hadoop, please use your favorite search engine; Hadoop will be mentioned throughout the book because this framework was the industry standard for big data processing for the past decade, and according to some people it still is. Many companies, especially in the Fortune 100, have a lot of infrastructure on that framework already. The Spark creators didn't want to reinvent the wheel, and they also wanted a better adoption rate for Spark in the companies. This is much easier on top of an existing infrastructure, so, for instance, Spark supports HDFS out of the box.

Let's have a look at the HDFS architecture. You might find it interesting because it's also a master-worker architecture:



*Figure 5: Overview of Master-Worker Hadoop Distributed File System Architecture*

The names of the nodes shown on the previous figure are neither master nor worker, but from the architecture it's pretty visible that the node "NameNode" is pretty important. Besides outlining the architecture, the previous picture is very important because it describes how distributed systems overcome failures. As systems get bigger, there is a pretty significant statistical chance that some of the nodes will experience outages from time to time. Data on data nodes have redundant copies on other data nodes in case a particular node fails, and so the master node knows where the backups are. If data nodes fail, data is simply read from copies. The master node is essentially a single point of failure in the HDFS; to cope with this, a backup node is added to the system. The backup node periodically checks if the master node is alive and syncs the data with it. If for some reason the name (master) node fails, the backup node will take over the master's role in the cluster.

A basic hello world example in big data analysis is a word count example, and we will go over how to do that with Spark in later chapters. Before going into the history of Spark and the basic concepts surrounding it, I would like to go over a very important data processing concept that has been around for quite some time now. It's a programming model called MapReduce. There are three basic phases in MapReduce. First is the Map, followed by a Shuffle phase, and the Reduce phase comes at the end. A basic overview is displayed in the following figure:

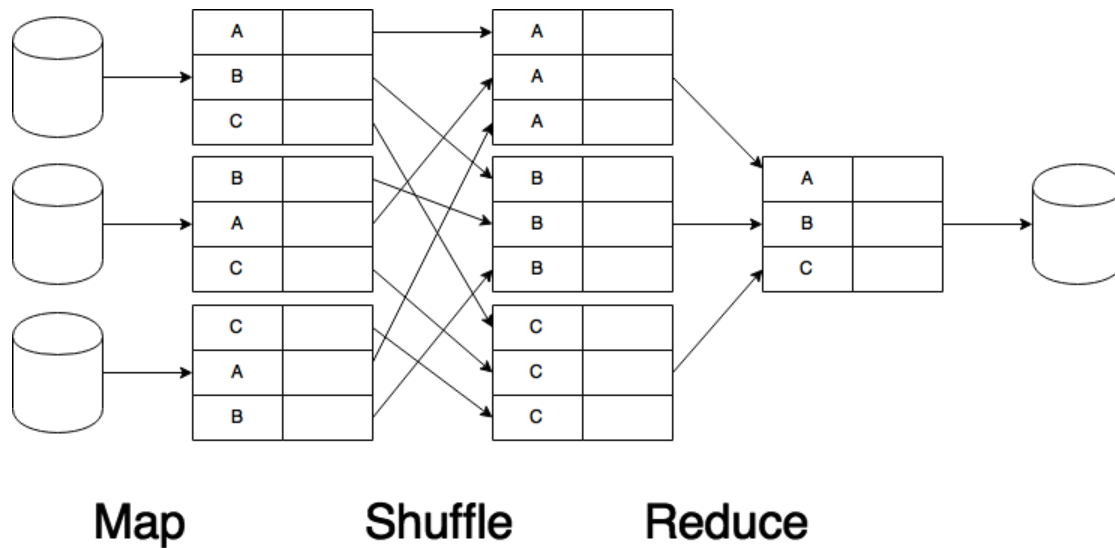


Figure 6: Overview of MapReduce Programming Model

The easiest way to describe MapReduce would be by using the word-count example. Imagine for a moment that we only have three words in a very large file. We are interested in the number of occurrences of each word. In the MapReduce programming model, every node would locally group the words together and make a count of every one of them in the map phase. The master node that is not shown on the previous figure would then assign every word to a specific node. The data is then shuffled in the next step so that it is associated with a specific node. The shuffle phase usually takes the longest because nodes have to transfer the data over the network and store it in their temporary storage. The node then combines info that came in from other nodes and produces a result—word count in our example—and stores it as a final step. At the end of every phase, data is stored to the disk. Writing the data to the disk is the part that usually takes the longest in data processing, and with MapReduce there are at least three phases where this takes place.

The basic concept behind the MapReduce model, used by Hadoop, is that it distributes the processing and disk load to nodes in the cluster; this was the industry standard when Spark was introduced in 2009. The main difference between Spark and Hadoop is that Spark can distribute the usage of not only the processor and the disk, but can also distribute in-memory operations. Because of that, it can achieve much greater speeds while processing data. For orientation, let's have a look at the times it takes to complete various computer related operations, from one CPU cycle up to a physical system reboot. To make it easier to compare, we'll add a human comparison column where the fastest operation will be mapped to a single second interval:

Table 1: Computer Task Processing Times Scaled to Human Time Perception

Event	Duration	Human comparison
1 CPU Cycle	0.3 ns	1 s
L1 cache access	0.9 ns	3 s
L2 cache access	2.8 ns	9 s

L3 cache access	12.9 ns	43 s
RAM access from CPU	120 ns	6 min
SSD I/O operation	50 – 150 us	2 – 6 days
Rotational disk I/O operation	1 – 10 ms	1 – 12 months
Internet: San Francisco to New York	40 ms	4 years
Internet: San Francisco to United Kingdom	81 ms	8 years
Internet: San Francisco to Australia	183 ms	19 years
TCP packet retransmit	1 – 3 s	105 – 317 years
Physical system reboot	5 min	32 millennia

The most important thing from the previous table for us is the comparison between RAM access from the CPU and rotational disk I/O operation. Once again, if one CPU cycle is a second, RAM I/O takes six minutes and rotational disk I/O operation takes somewhere around one month or even a whole year. Modern computer systems have significant amounts of RAM available to them, so it was probably only a question of time when somebody was going to figure out that RAM can be used as a distributed resource to process large amounts of data. This is actually one of Spark's first killer features, and we will describe the details in the chapters to come. Let's have a look at how Spark developed in the next section.

## A Brief History of Spark

Spark was created at UC Berkley AMPLab in 2009 by Matei Zaharia. It might sound unbelievable, but the first version of Spark was written in only 1600 lines of Scala code. One of the main goals behind Spark was to make a big data processing framework that is fast enough for machine learning. Hadoop was not usable for this approach because there was a lot of disk usage involved, and near-real time data processing took around twenty to thirty minutes at the time Spark was created. In its initial version, Spark could solve the same tasks in under a minute because of aggressive memory usage. The Spark project was open-sourced in 2010 under a BSD license. The BSD license allows unlimited redistribution for any purpose as long as its copyright notices and the license's disclaimers of warranty are maintained. The license also contains a clause restricting use of the names of contributors for endorsement of a derived work without specific permission. In 2013, the project was donated to the Apache Software Foundation. It also switched its license to Apache 2.0. The Apache license is a bit more restrictive when it comes to copyright and redistributing code and also has special requirements, mostly pertaining to giving proper credit to those who have worked on the code and to maintaining the same license. In the same year, the creators of Spark started a company called Databricks. The company's goal was to help clients with cloud-based big data processing by using Spark. In February 2014 Spark became a top-level Apache project.

The next very important thing happened in November 2014. Spark won the Daytona GraySort contest. Daytona GraySort is a competition where various companies come to show off their big data processing frameworks and solutions. The basic goal is to sort 100 terabytes of data (consisting of one trillion records) as fast as possible. The data that needs to be sorted is located on the HDFS, the inner workings of which we described in the previous section. Sorting the data usually takes around 500 TB of disk I/O and around 200 TB of network I/O. Organizations from around the world often build dedicated sort machines with specialized hardware and software. Winning the 2014 sort competition is a very important milestone for the Spark project, especially when the previous world record set by Yahoo with Hadoop MapReduce is taken into account:

*Table 2: Daytona GraySort 2014 Competition Results*

	<b>2013 Record: Hadoop</b>	<b>2014 Record: Spark</b>
Data Size	102.5 TB	100 TB
Elapsed Time	72 minutes	23 minutes
Number of Nodes	2100	206
Sort rate	1.42 TB/min	4.27 TB/min
Sort rate per node	0.67 GB/min	20.7 GB/min

The amount by which the previous record is surpassed is simply incredible. Spark actually managed to process the data three times faster with ten times fewer machines.

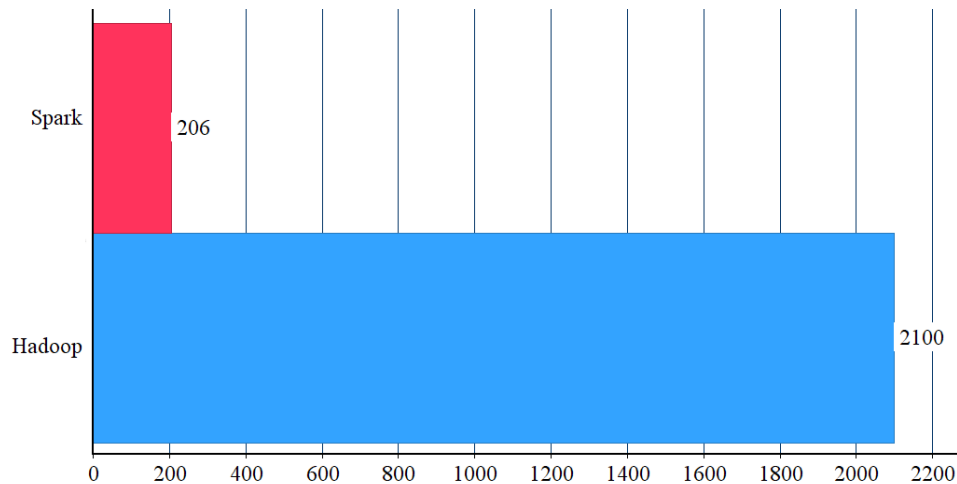


Figure 7: Number of Nodes Used in the GraySort Challenge by Spark and Hadoop

The results of this competition attracted a lot of developers from around the world, and Spark had over 465 contributors in 2014. This makes it the most active project in the Apache Software Foundation and probably the most active big data open source project. In the next section we'll make an overview of Spark as a platform for big data processing.

## Spark Overview

Spark consists of multiple components. The central Spark component is Spark Core. All of the other components have this component in common. This approach has many benefits. One of them is when optimization is added to the core, all of the other components start to use this optimization right away. The other advantage is that the code base of Spark remains compact and reinventing the wheel is reduced to a minimum. A Spark overview is shown in the following figure:

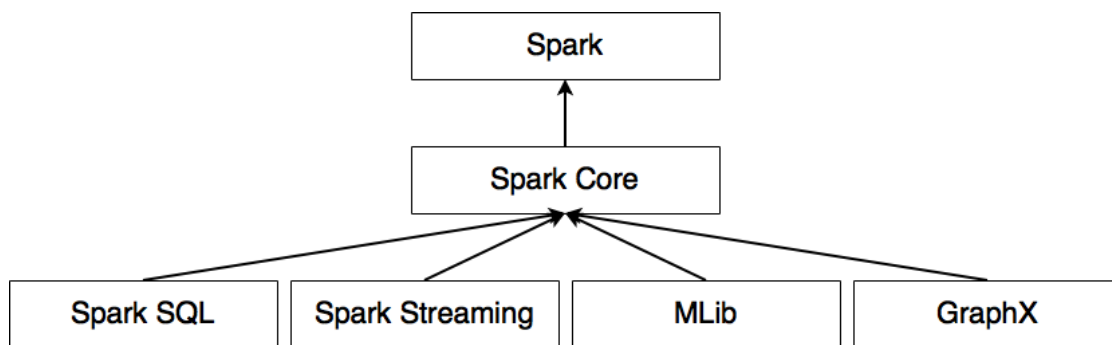


Figure 8: Spark Components

Spark Core is in charge of memory management, task scheduling, error recovery and storage systems interaction. It also defines the resilient distributed datasets. We'll talk about them in chapters to come, but for now it is sufficient to describe them as distributed collections of items that can be manipulated in parallel.



Spark SQL is a package for working with structured data. The component started out as a support for Hive Query Language, but over time it grew to be a component that supports working with almost any kind of data, from JSON to large datasets residing on distributed storage. Spark SQL gives developers a unique ability to combine programming constructs together with SQL-like syntax when working with structured data.

Spark Streaming enables processing of data streams. Some of you might wonder what a data stream is. In essence, it's a continuous influx of data from all sorts of sources, like log files or queuing solutions that take in messages from the clients. The traditional big data processing frameworks are oriented towards batch processing of data. This component from Spark is actually a step towards modern data processing frameworks because it's designed from the beginning to work with live, incoming data and to generate the results on the fly.

MLlib provides functionalities for machine learning. In essence, it's a loose collection of high-level algorithms. One of the main driving forces for Spark's creation was machine learning. Solutions available at the time were relatively slow because they didn't exploit the speed of RAM. In the past decade, machine learning at this scale was based around a library called Mahout, and it was primarily used in Hadoop ecosystems. In 2014, Mahout announced it would no longer accept Hadoop MapReduce code, and it switched new development to Spark's library MLlib.

GraphX is a library for graph data manipulation. It allows users to create and manipulate graph data consisting of a vertex and edges, and with it we complete a short overview of Spark. In the next chapter we are going to go over Spark installation steps.

# Chapter 1 Installing Spark

Spark runs in the Java Virtual Machine. The prerequisite for running Spark is installing Java. We are going to go over the installation steps on the most popular operating systems, like Microsoft Windows and Linux. Since release 1.4, Spark also supports the R programming language and Python 3. The previous versions of Spark only supported Python 2.6. The R language is mentioned for completeness only; this book is not going to go deeper into R language examples beyond stating that Spark has built-in support for it.

## Installing Spark Prerequisites on Linux

We are going to demonstrate how to install Apache Spark on Centos 6.X from scratch. As mentioned earlier, Spark actually runs on top of JVM, so we have to install Java first. Most of the desktop versions of Linux systems come with Java preinstalled, but we will cover Java installation just in case. Note that the installation in the examples was done with the user having root privileges. You might need to run all of the following commands in this section by prefixing them with **sudo** in order to acquire root privileges.

### Installing Java

Skip this step if your system has an appropriate version of Java installed. The easiest way to check this is by running the command **java -version** in the shell. While Java version 8 has been available for quite some time now, many developers work on Java 7 version. You can install whichever Java version you prefer. The provided source codes will be in both Java 7 and Java 8 versions.

*Code Listing 1: Installing Java on Linux*

```
$ cd /opt/

$ wget --no-check-certificate -c --header "Cookie: oraclelicense=accept-securebackup-cookie"
http://download.oracle.com/otn-pub/java/jdk/8u51-b16/jdk-8u51-linux-x64.tar.gz

$ tar xzf jdk-8u51-linux-x64.tar.gz

$ cd jdk1.8.0_51/

$ alternatives --install /usr/bin/java java /opt/jdk1.8.0_51/jre/bin/java 2
$ alternatives --install /usr/bin/javaws javaws /opt/jdk1.8.0_51/jre/bin/javaws 2
$ alternatives --install /usr/bin/javac javac /opt/jdk1.8.0_51/bin/javac 2
$ alternatives --config java
```

```
There is 1 program that provides 'java'.
Selection  Command
-----
*+ 1      /opt/jdk1.8.0_51/jre/bin/java
Enter to keep the current selection[+], or type selection number: 1 [ENTER]

$ java -version
java version "1.8.0_51"
Java(TM) SE Runtime Environment (build 1.8.0_51-b16)
Java HotSpot(TM) 64-Bit Server VM (build 25.51-b03, mixed mode)

[Add these to ~/.bash_profile or the exports will not be set on the next boot]
$ export JAVA_HOME=/opt/jdk1.8.0_51
$ export JRE_HOME=/opt/jdk1.8.0_51/jre
$ export PATH=$PATH:/opt/jdk1.8.0_51/bin:/opt/jdk1.8.0_51/jre/bin
```

## Installing Python

If you are running Linux, there is a fair chance that you already have Python installed. You can check this by running the following commands:

*Code Listing 2: Checking Python Version on Linux*

```
$ python -V
Python 2.6.6
```

If you are not satisfied with an installed version of Python, you can install a newer one. It's a good practice to update OS packages before starting the installation.

*Code Listing 3: Updating Linux Packages by Using Yum*

```
$ yum -y update
```

Installing multiple versions from the Python source can quickly become cumbersome. I would recommend you use the pyenv tool. The prerequisite for installing pyenv is a popular source versioning system Git. After pyenv is installed, it will automatically build the downloaded Python from the source. To run the build processes successfully, you also need to install development tools and some other components required by Python. You can install the components by running the following commands:

*Code Listing 4: Installing Git and Development tools by Using Yum*

```
$ yum install git

$ yum groupinstall "Development tools"
```

```
$ yum install bzip2-devel
$ yum install openssl-devel
```

*Code Listing 5: Installing Pyenv*

```
$ curl -L https://raw.githubusercontent.com/yyuu/pyenv-installer/master/bin/pyenv-installer |
bash

# add following to ~/.bash_profile:

export PATH="$HOME/.pyenv/bin:$PATH"
eval "$(pyenv init -)"
eval "$(pyenv virtualenv-init -)"
```

After installing pyenv, switching between Python versions is relatively easy even on the project level. All you need to do is install the Python version that you need and then create a **.python-version** file inside the project directory with the Python version that you need:

*Code Listing 6: Installing Pyenv and Switching Python Version*

```
$ pyenv install 3.4.3

# version 3.4.4 is installed but if you try to run the version command you get something like

$ python -V
Python 2.6.6

# now create a file .python-version and put 3.4.3 in it, if you now run version command

$ python -V
Python 3.4.3
```

## Installing Scala

Spark is pretty flexible when it comes to Java and Python, but versions of Scala must match. At the time of writing, the latest Spark version is 1.4.1 and is compatible with Scala 2.10.x. Be sure to check the latest Spark documentation when you start to install Scala on your system. In my case, it's going to look like the following:

*Code Listing 7: Installing Scala on CentOS*

```
$ wget http://www.scala-lang.org/files/archive/scala-2.10.5.tgz

$ tar xvf scala-2.10.5.tgz

$ mv scala-2.10.5 /usr/lib
$ ln -s /usr/lib/scala-2.10.5 /usr/lib/scala

# you can add following commands to ~/.bash_profile:
$ export PATH=$PATH:/usr/lib/scala/bin
$ export SCALA_HOME=/usr/lib/scala

# check if everything is o.k. by running
$ scala -version
Scala code runner version 2.10.5 -- Copyright 2002-2013, LAMP/EPFL
```

To build applications with Scala, you will also need to use a tool called Simple Build Tool. To install it use the following commands:

*Code Listing 8: Installing sbt on CentOS*

```
$ curl https://bintray.com/sbt/rpm/rpm | sudo tee /etc/yum.repos.d/bintray-sbt-rpm.repo

$ yum install sbt

$ sbt about
[info] This is sbt 0.13.8
```

## Installing Spark Prerequisites on Windows

Windows 7 simply shows no signs of dying and is still by far the most popular Windows OS around. It doesn't have mainstream support from Microsoft, but that hasn't changed its market share significantly. The installation procedure will be described based on Windows 7.

## Installing Java

For Windows, go to <http://www.oracle.com/technetwork/java/javase/downloads/index.html> and locate the JDK download. At the time of writing the latest version is 8u51. Agree to the terms and conditions on the page to download the Java installer and run the downloaded software. Just make sure that you select a JDK for Windows and be careful about 32(x86) and 64(x64) bit systems. Before proceeding with the installation, Windows will ask you if you really want to make changes to your computer:

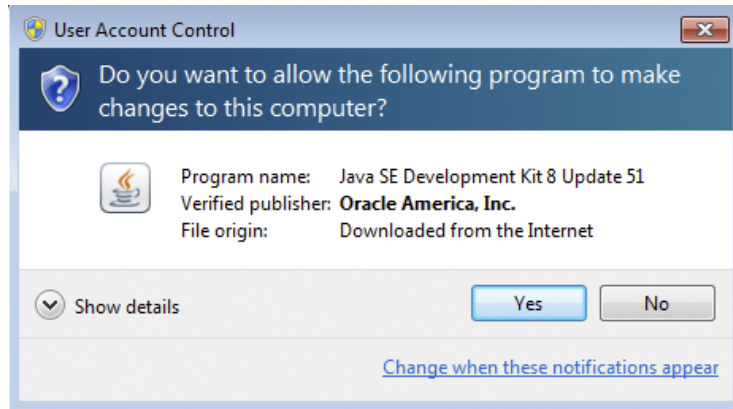


Figure 9: Allow the downloaded installer to make changes to the system.

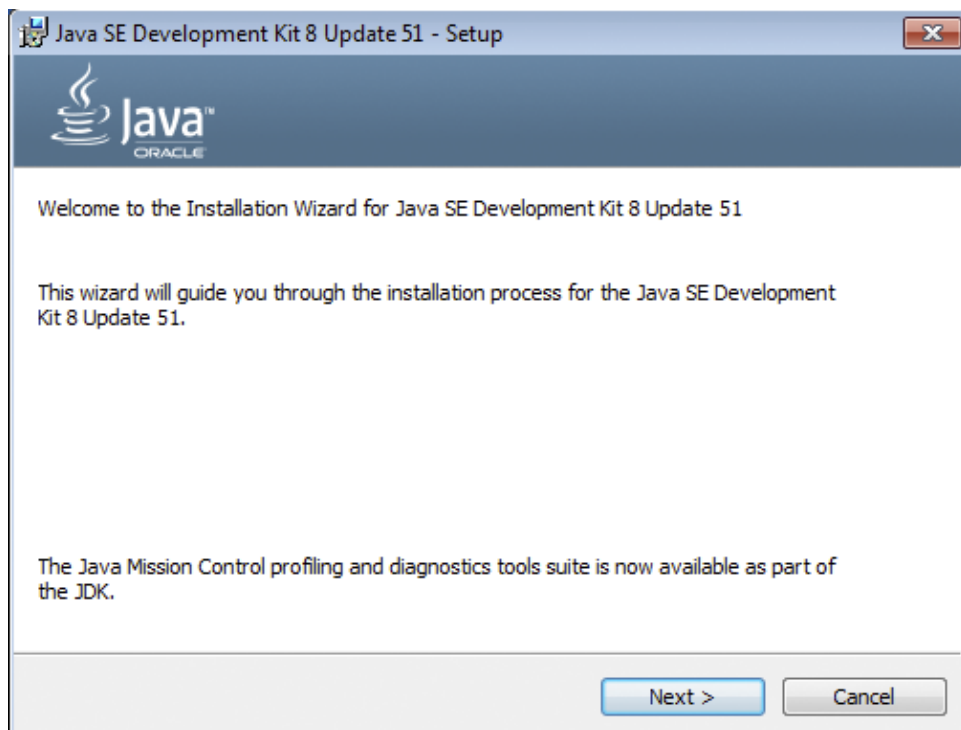


Figure 10: Initial Screen for Installing JDK on Windows



*Figure 11: The End of a Successful JDK Install*

After installing Java, it is important to set the **JAVA\_HOME** environment variable. Use the following steps and check or set the **JAVA\_HOME** variable:

1. Right-click on **My Computer**.
2. Select **Properties** from the list.
3. Go to **Advanced system settings**. This will open **System Properties** on the **Advanced** tab.
4. Click on **Environment Variables**.
5. Add or update the **JAVA\_HOME** variable as shown on the following figure. The default location is something like **C:\Program Files\Java\jdk1.8.0\_51**; make sure that you check the location on file system before changing the variable. Note that the location varies depending on the installed Java version.

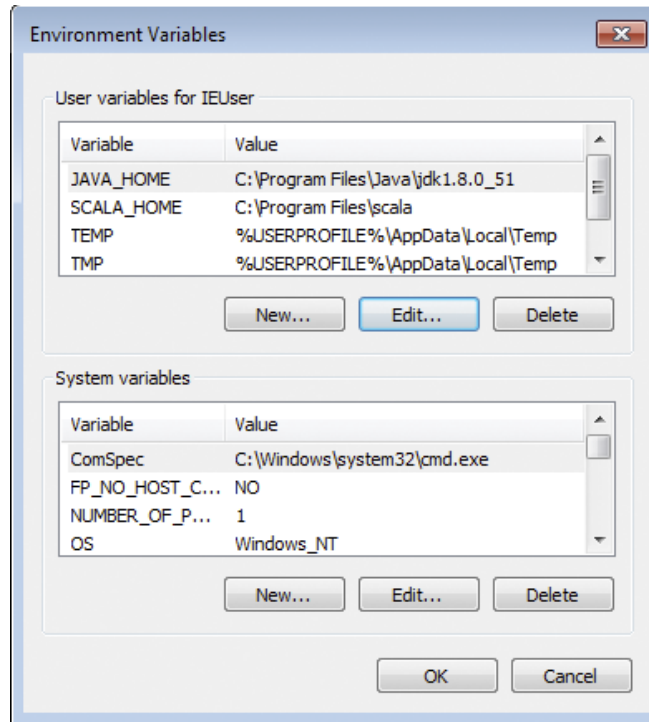


Figure 12: Java Home Environment Variable Setup

## Installing Python

Many Python developers don't want to move to the new Python 3 syntax, so Python has both Python 2 and 3 available for download. Go to <https://www.python.org/downloads/windows/> and select the one you like the most. Note that there are x86 and x86-64 versions available, so please make sure whether you have a 32 or 64 bit system before downloading the installer.

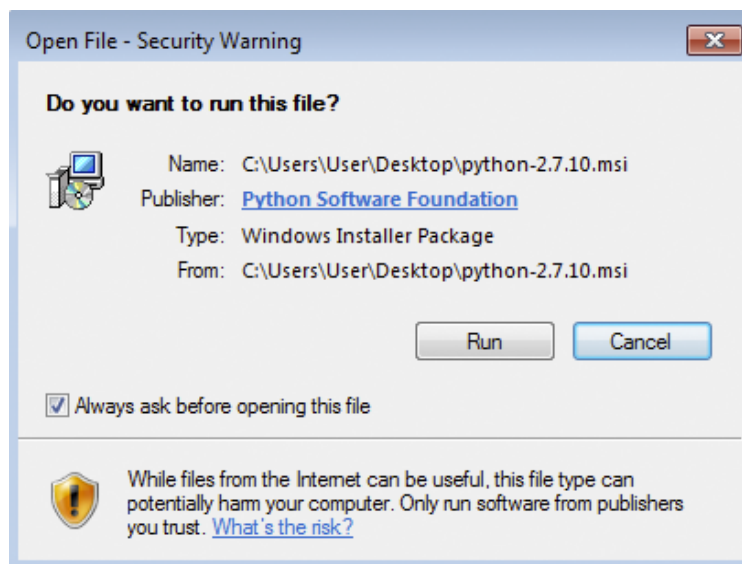


Figure 13: Click on run when Windows security popup comes up.





Figure 14: Install Python for all users.

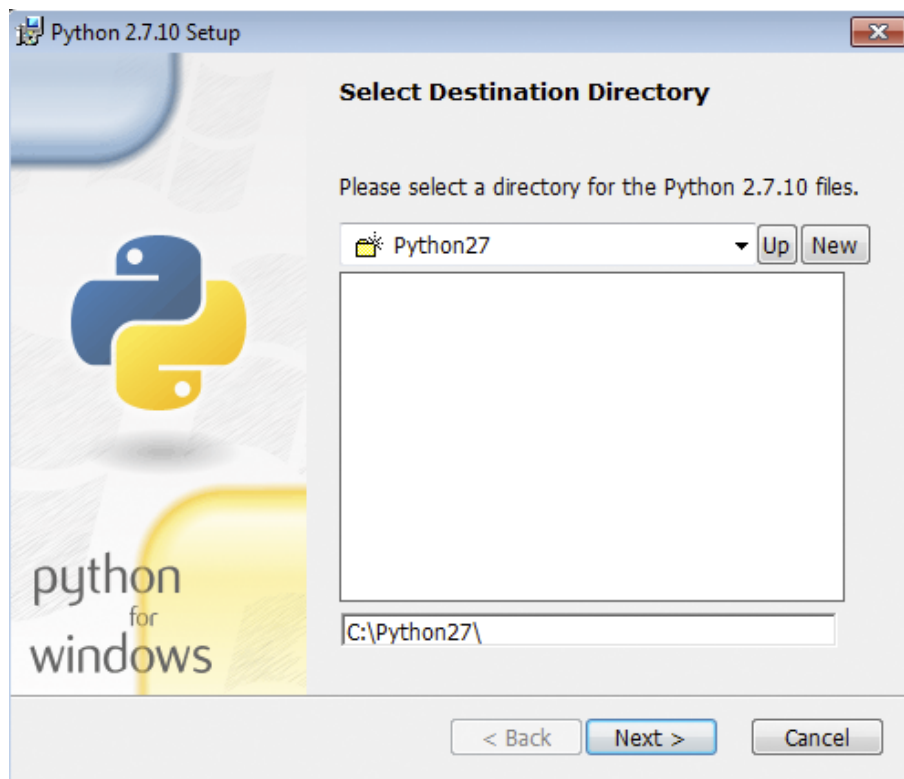


Figure 15: Select destination directory.

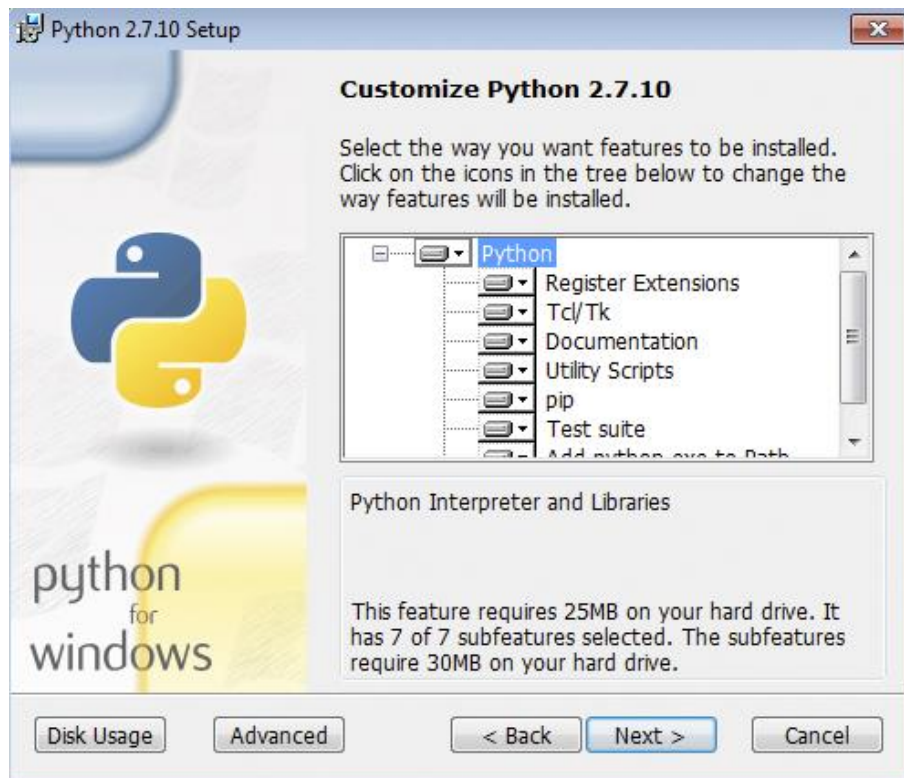


Figure 16: Click on Python in dropdown, choose Entire feature will be installed.



Figure 17: End of Python Installation

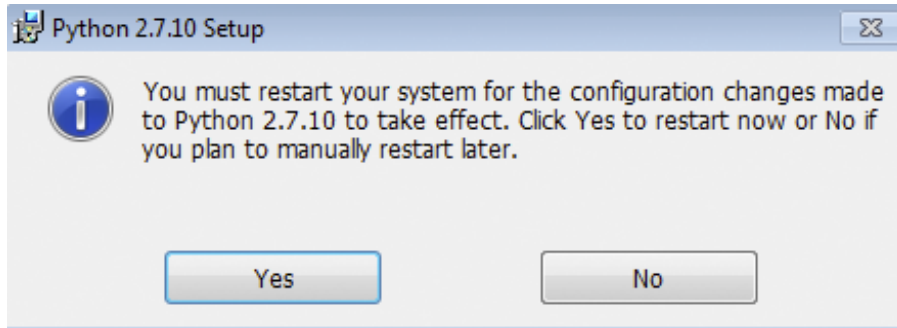


Figure 18: You may need to restart your system before using Python.

## Installing Scala

At the time of writing, the latest Spark version is 1.4.1 and is compatible with Scala 2.10.x. Check the latest Spark documentation before you start to install Scala on your system. To install Scala on Windows, go to <http://www.scala-lang.org/download/all.html> and find the appropriate version to download. On my system with the latest Spark version, it's <http://www.scala-lang.org/download/2.10.5.html>. Installation steps are described in the following figures; if there are any security questions or something similar, agree with them:

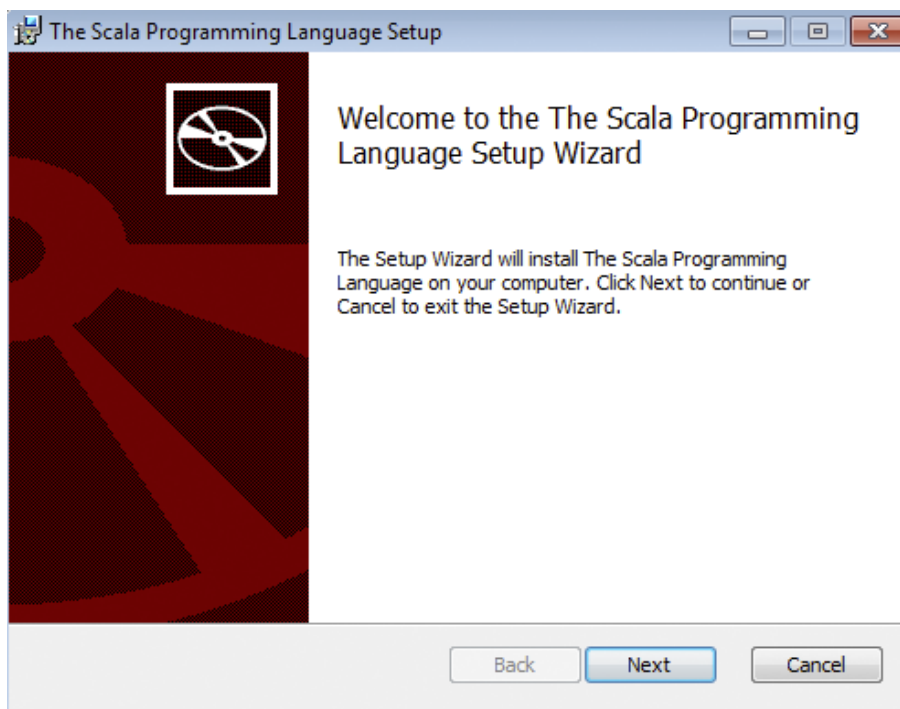


Figure 19: On the initial Scala install screen, click Next.

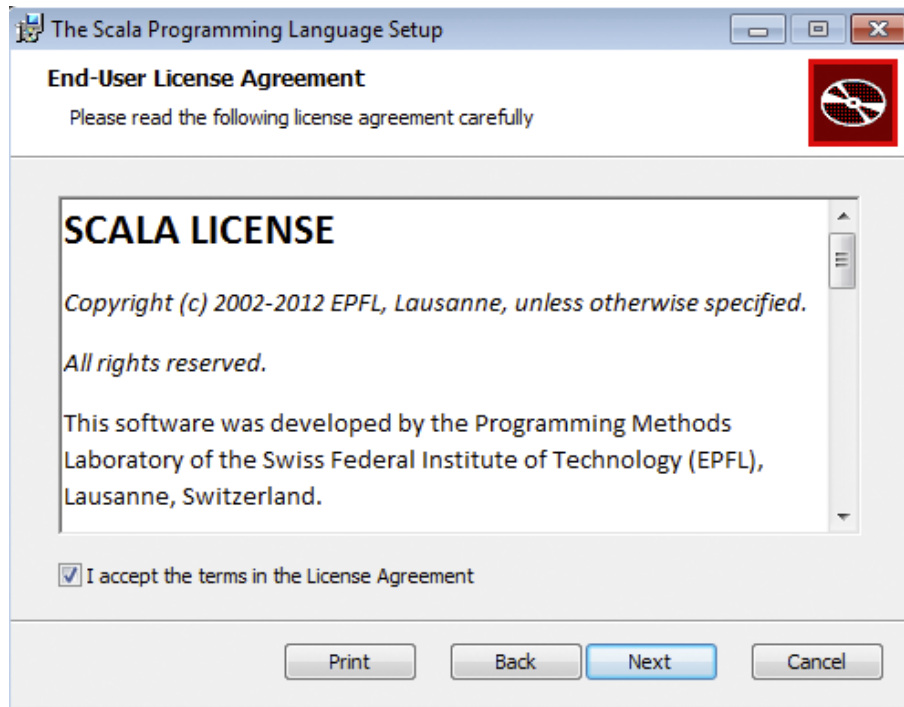


Figure 20: Check the license and accept the terms.

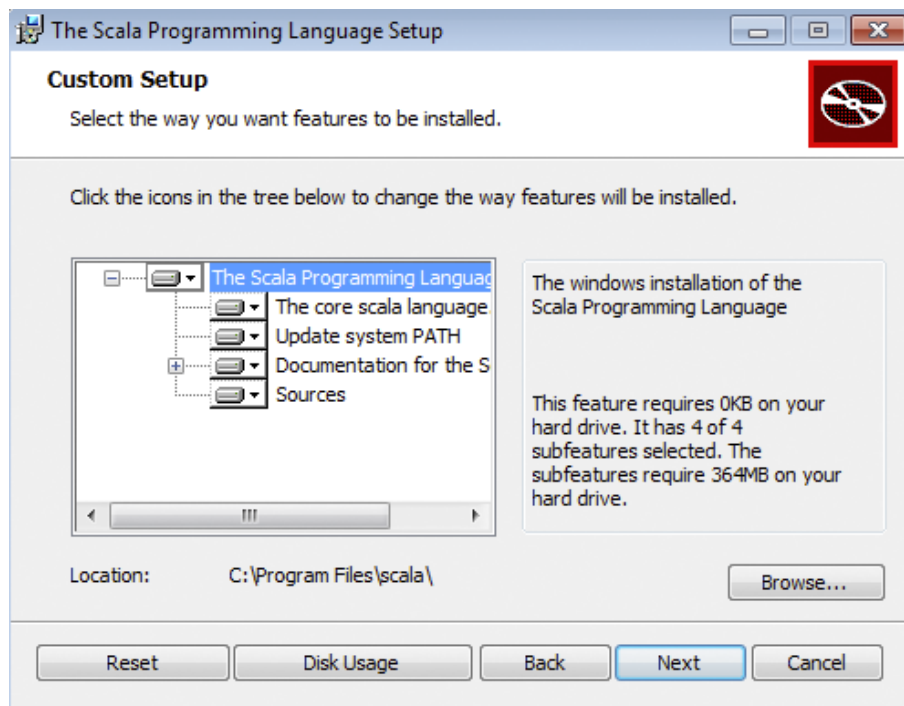
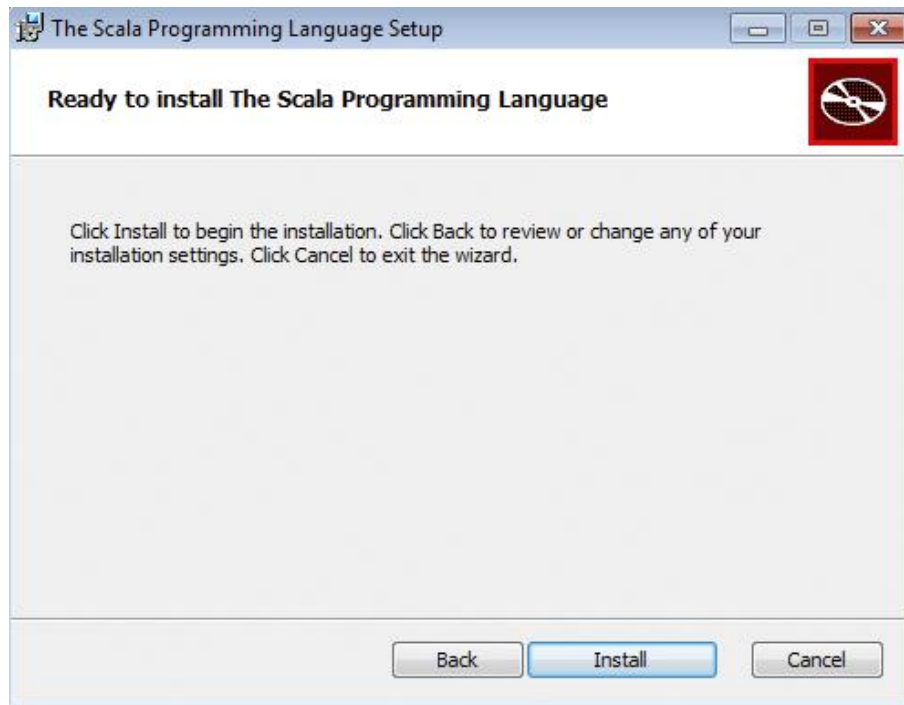
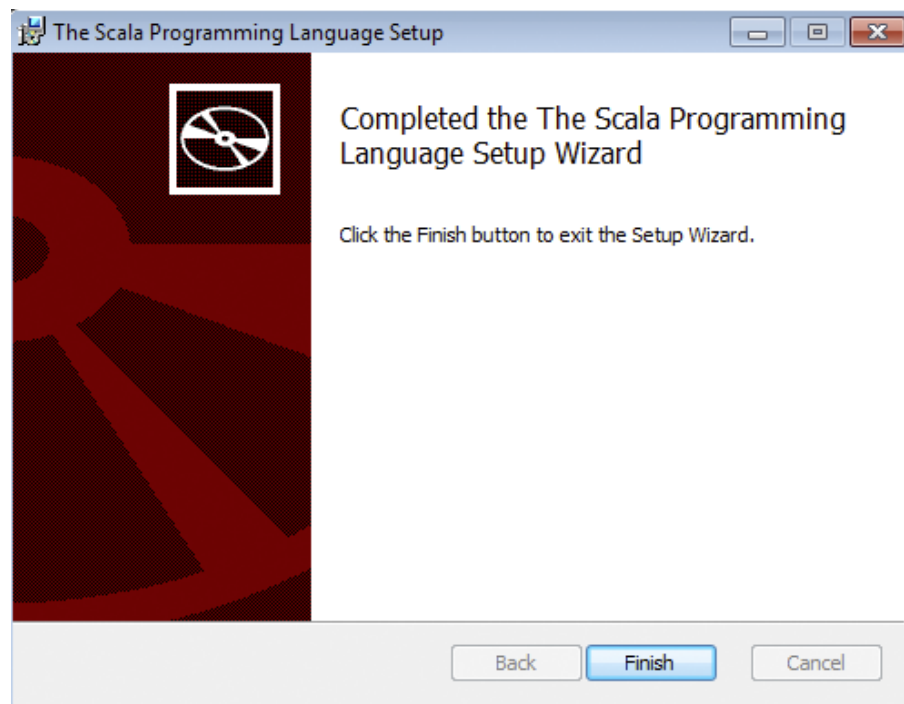


Figure 21: Install all features on local disk and click Next.



*Figure 22: Start the installation process.*

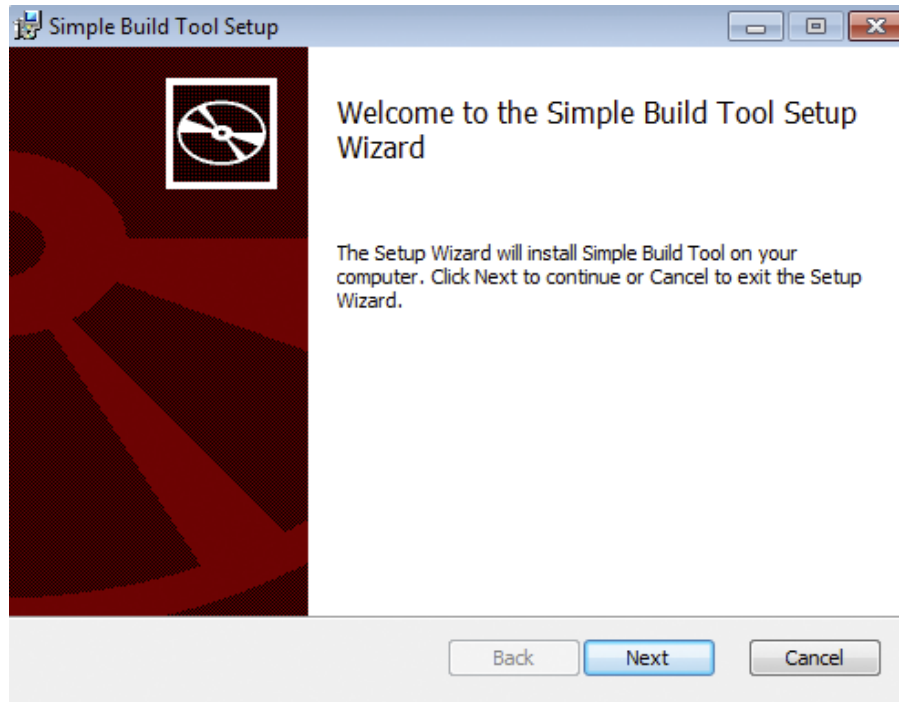


*Figure 23: End of Scala Installation Process*

At the end of the installation process, set environment variable **SCALA\_HOME** to value "C:\Program Files\scala." Use the process described in the previous section on Java.

If you want to develop applications for Spark in Scala, you will need to install the Simple Build Tool.

You can download the Simple Build Tool for Scala at <http://www.scala-sbt.org/download.html>. After you download it, run the installer:



*Figure 24: Start of sbt Install Process*



Figure 25: Accept the license terms.

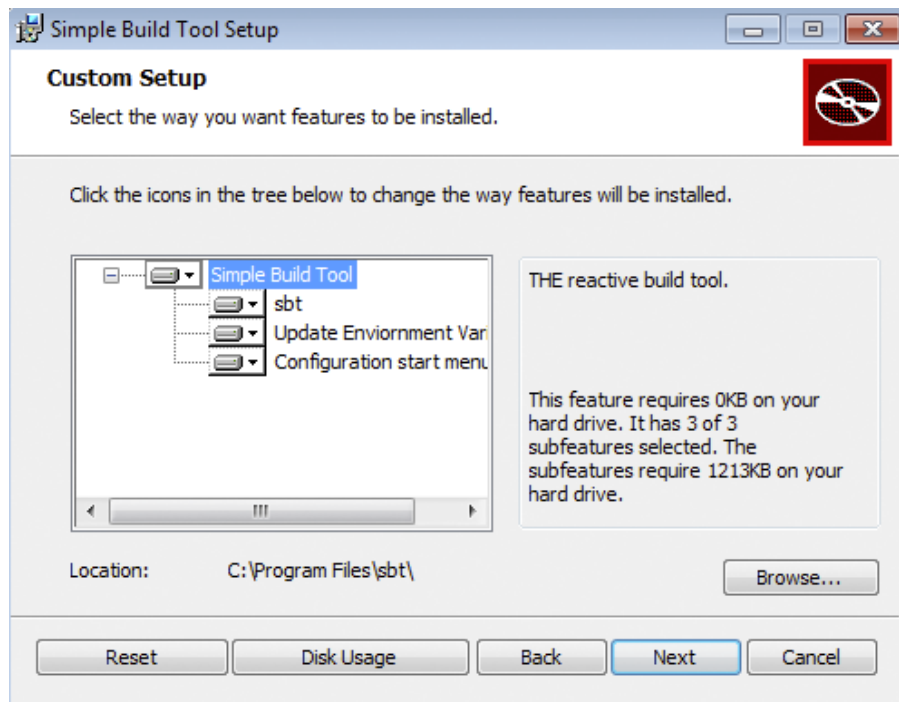
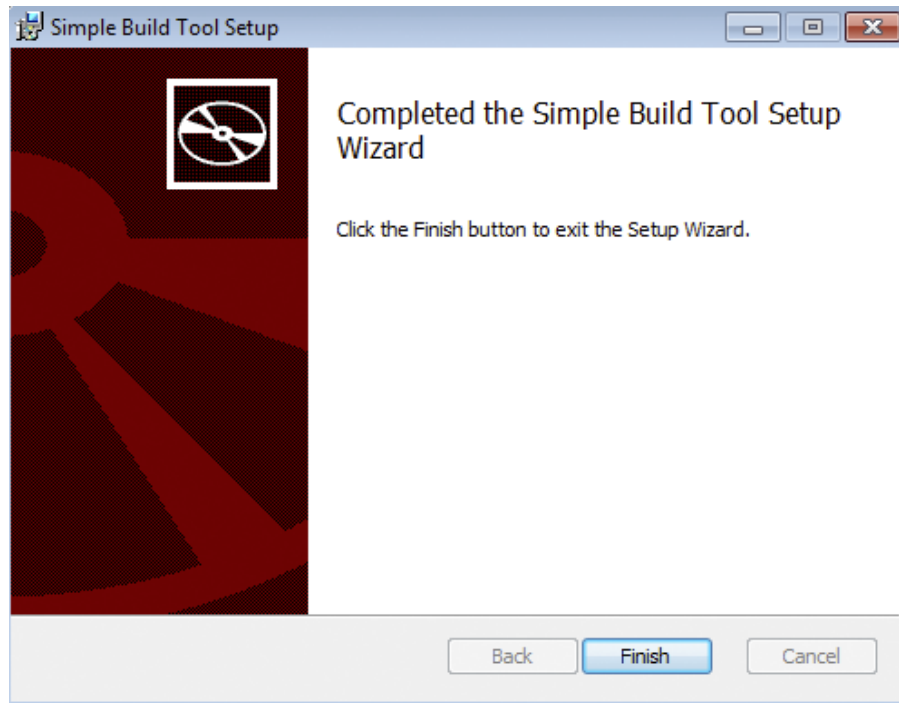


Figure 26: Install all features to the hard drive in the Simple Build Tool dropdown menu.





*Figure 27: Last Screen During the Installation of Simple Build Tool*

With this, all prerequisites for the rest of the book, including how to install Spark, are covered. The installation and configuration process is almost the same on both Windows and Linux from now on.

## Spark Download and Configuration

You can download Spark at <https://spark.apache.org/downloads.html>. At the moment, the latest version is 1.4.1. Pick this one when choosing a Spark release on the download page. If you plan to run Spark on existing Hadoop infrastructure, choose your Hadoop version under the package type selection. If you don't intend to use Hadoop, simply select "pre-built for Hadoop 2.6 and later." Hadoop version in download type doesn't really matter for this book. The next element on page will be a download link; some users may be surprised that there is no download button. Download the archive to a directory of your choosing. In my case, the download link is <http://www.apache.org/dyn/closer.cgi/spark/spark-1.4.1/spark-1.4.1-bin-hadoop2.6.tgz>, but this may vary depending on whether there is a newer version of Spark or Hadoop when you start the download. You might have noticed the archive is in "tar" format. Users of Linux computers will have no special trouble unpacking the downloaded file, but on Windows you will need to install a utility like 7-Zip or any other archiving software that has tar support. Note that the extraction process will have multiple steps because there are archives within archives. When extracting the output files, always output them to a folder with a simple name. I ended up with something like "C:\spark\spark\spark\spark-1.4.1-bin-hadoop2" on Windows, but then I simply copied the contents of the final folder to "C:\spark-1.4.1" to make later interactions with Spark easier. On Linux, I copied and extracted Spark to the folder "/root/spark-1.4.1-bin-hadoop2.6;" it really depends on your preferences.





**Tip:** Extract Spark's *tgz* archive to a folder with a path that has no spaces in it.

If you are running Linux, open your shell and navigate to the folder where you extracted Spark. If you are running Windows, open the Command Prompt and do the same thing. First we'll run the Spark Shell.

*Code Listing 9: Running Spark Shell from Command Prompt on Windows*

```
C:\spark-1.4.1>bin\spark-shell
```

*Code Listing 10: Running Spark Shell from Shell on Linux*

```
[root@localhost spark-1.4.1-bin-hadoop2.6]# ./bin/spark-shell
```

The result will be a running Spark Shell on both platforms. If everything worked, you should see a read-eval-print loop scala interface:

*Code Listing 11: Result of previous commands should be a running scala interpreter interface.*

```
scala>
```

I personally don't like the amount of information that is printed when running Spark Shell with defaults. It's really low-level info that has very little use. Although you just started the Spark Shell, it might be a good idea to leave it and change the logging level so that it doesn't confuse you. To leave the REPL simply type **exit()** and hit the **Enter** key.

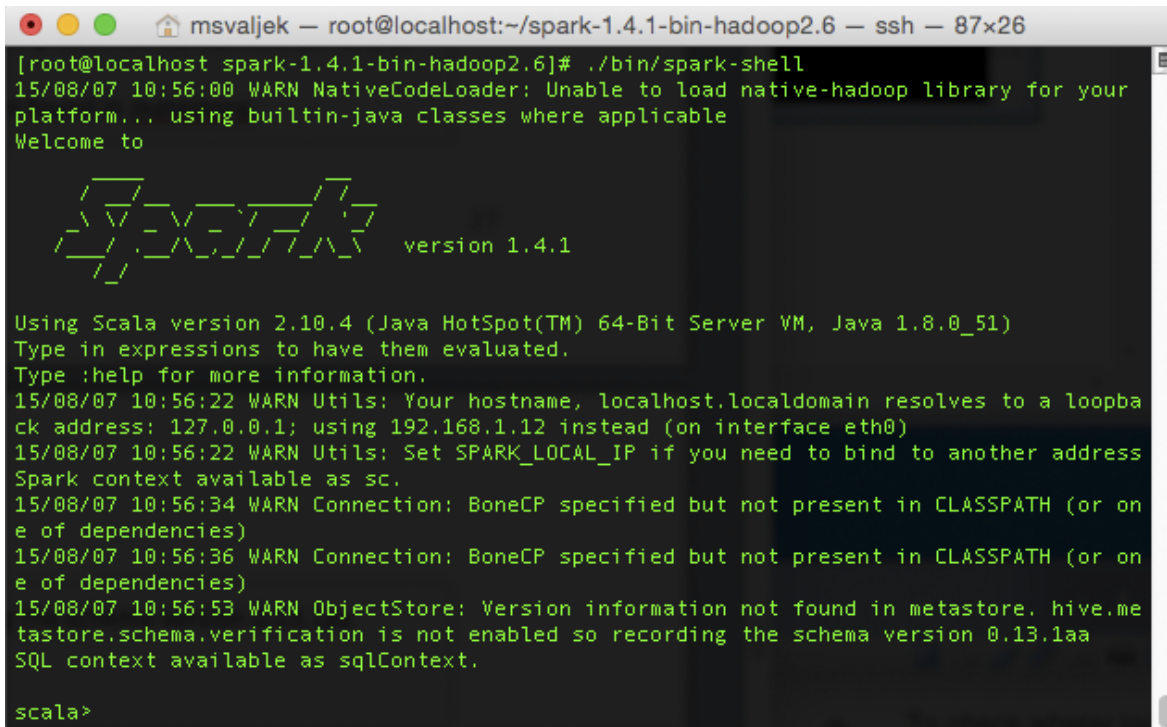
Navigate to the **conf** subfolder and make a copy of the file **log4j.properties.template**. Rename it **log4j.properties** and open the file for editing. Change the line **log4j.rootCategory** from **INFO** to **WARN** as shown in following code listing:

*Code Listing 12: More User Friendly Logging Setup*

```
# Set everything to be logged to the console
log4j.rootCategory=WARN, console
log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender.console.target=System.err
log4j.appender.console.layout=org.apache.log4j.PatternLayout
log4j.appender.console.layout.ConversionPattern=%d{yy/MM/dd HH:mm:ss} %p %c{1}: %m%n

# Settings to quiet third-party logs that are too verbose
log4j.logger.org.spark-project.jetty=WARN
log4j.logger.org.spark-project.jetty.util.component.AbstractLifeCycle=ERROR
log4j.logger.org.apache.spark.repl.SparkIMain$exprTyper=INFO
log4j.logger.org.apache.spark.repl.SparkILoop$SparkILoopInterpreter=INFO
```

After setting the log output level to Warning, change your current directory to one level up from the **conf** folder and start the Spark shell again. You should see a lot less output coming out from the shell before it gives you the scala REPL:



```
[root@localhost spark-1.4.1-bin-hadoop2.6]# ./bin/spark-shell
15/08/07 10:56:00 WARN NativeCodeLoader: Unable to load native-hadoop library for your
platform... using builtin-java classes where applicable
Welcome to

      /_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_\
     /_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_\
    /_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_\
   /_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_\
  /_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_\
 /_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_\
/_/_/_/_/_/_/_/_/_/_/_/_/_/_/__\

version 1.4.1

Using Scala version 2.10.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_51)
Type in expressions to have them evaluated.
Type :help for more information.
15/08/07 10:56:22 WARN Utils: Your hostname, localhost.localdomain resolves to a loopba
ck address: 127.0.0.1; using 192.168.1.12 instead (on interface eth0)
15/08/07 10:56:22 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another address
Spark context available as sc.
15/08/07 10:56:34 WARN Connection: BoneCP specified but not present in CLASSPATH (or on
e of dependencies)
15/08/07 10:56:36 WARN Connection: BoneCP specified but not present in CLASSPATH (or on
e of dependencies)
15/08/07 10:56:53 WARN ObjectStore: Version information not found in metastore. hive.me
tastore.schema.version is not enabled so recording the schema version 0.13.1aa
SQL context available as sqlContext.

scala>
```

*Figure 28: Spark Shell with Logging Level Set to Warn*

This concludes the chapter. You now have a running Spark instance on your system. In the next chapter, we are going to go over how to start data processing tasks in Scala, Java, and Python.

# Chapter 2 Hello Spark

The hello world applications of big data processing always involve searching for word occurrences. Usually the data for processing comes in from a stream, database, or distributed file system. The examples in this chapter are on a hello world level, so we are simply going to use files provided with Spark. Here is a list of files within the Spark directory:

*Code Listing 13: Contents of Spark Directory*

```
.
├── CHANGES.txt
├── LICENSE
├── NOTICE
├── R
├── README.md ←
├── RELEASE
├── bin
├── conf
├── data
├── ec2
├── examples
├── lib
├── python
└── sbin
```

In this chapter, we are going to use the **README.md** file in our examples. Spark was written in Scala, so throughout the book we are going to start with Scala examples first. If you are interested in other programming languages, you can skip to the example that you need or like. I'm not really an R expert, so I won't provide examples throughout the book for it. I also didn't cover installation of R in previous chapters. At the moment, Java has no interactive shell support like Python and Scala do, so when we come to Java examples we are going to demonstrate how to set up a basic Java Spark project. Dependency management for Java examples can become pretty complex pretty fast, so we'll also cover how to manage dependencies with the most popular dependency tool for Java today, Maven. We'll start with a simple text search.

## Counting Text Lines Containing Text

In this example, we are going to count lines of text containing certain words. Although this might not seem like much, this is already pretty useful. For instance, you could count how often an error appeared in the production log or something similar. In the introduction we mentioned that for simplicity's sake we are just going to use the **README.md** file that comes with the Spark installation. Let's begin with Scala.

## Scala Example

Open the Command Prompt (Windows) or Shell (Linux) and go to the directory where you unpacked the Spark installation. The previous chapter describes how you can do this, but as a short refresher here is how:

*Code Listing 14: Running Spark Shell from Command Prompt on Windows*

```
C:\spark-1.4.1>bin\spark-shell
```

*Code Listing 15: Running Spark Shell from Shell on Linux*

```
[root@localhost spark-1.4.1-bin-hadoop2.6]# ./bin/spark-shell
```

The Advantage of using Spark Shell is that you have Spark Context available to you right away, and you don't have to initialize it. We'll cover how to do that in chapters to come, but for now it's sufficient to say that Spark Context represents a connection to the Spark cluster and is used to do computational operations on it. The context available in the shell is not actually an out-of-the-box connection to a cluster, it's a context intended for testing, evaluations, and runs on a single node or in the development phase. Luckily, that is just what we need at the moment. Enter the following:

*Code Listing 16: Counting Lines Containing Word Spark in Scala*

```
val readmeFile = sc.textFile("README.md")
val sparkMentions = readmeFile.filter(line => line.contains("Spark"))

// count the lines having Spark in them
sparkMentions.count()
```

If everything went fine, you should see something like this:

*Code Listing 17: Result of Counting Lines in Spark Shell*

```
scala> sparkMentions.count()
res2: Long = 19
```

The `textFile` method on Spark Context turns a file into a collection of lines. After that, we will filter out the lines that don't have Spark in them. In the end we will be calling an action called `count`. It's all pretty straightforward. If you know a little bit more about Spark, you know that the `readmeFile` is not just any collection type like array or list, and that in Spark this collection is actually called *resilient distributed dataset* and that this type of collection has a lot of methods for various transformations and actions. We'll come to that topic soon. For now, it's sufficient that you think about the results as if they were collections of items.

From time to time you will also want to check if everything is fine with the files that you are reading just to make sure you can very easily have a peek into the file. For instance, if you wanted to print the first ten lines in a file, you would do the following:

*Code Listing 18: Peek into File by Taking First Ten lines*

```
scala> readmeFile.take(10)

res5: Array[String] = Array(# Apache Spark, "", Spark is a fast and general cluster computing system
for big data. It provides, high-level APIs in Scala, Java, and Python, and an optimized engine that,
supports general computation graphs for data analysis. It also supports a, rich set of higher-level
tools including Spark SQL for SQL and structured, data processing, MLlib for machine learning,
GraphX for graph processing,, and Spark Streaming for stream processing., "",
<http://spark.apache.org/>)
```



**Tip:** *Make sure you have the right file by using take.*

The lines count example is case sensitive, so only the lines containing Spark written with capital case will be counted. But it's enough to get you going. In the next section we are going to go over lines containing a string example from Python's angle.

## Python Example

To start using Spark with Python, we will use the PySpark utility. Running it is pretty much the same as running Spark Shell from the previous section:

*Code Listing 19: Running Spark Shell from Command Prompt on Windows*

```
C:\spark-1.4.1>bin\PySpark
```

*Code Listing 20: Running Spark Shell from Shell on Linux*

```
[root@localhost spark-1.4.1-bin-hadoop2.6]# ./bin/PySpark
```

The result of the previous commands should be a Python Spark shell as shown here:



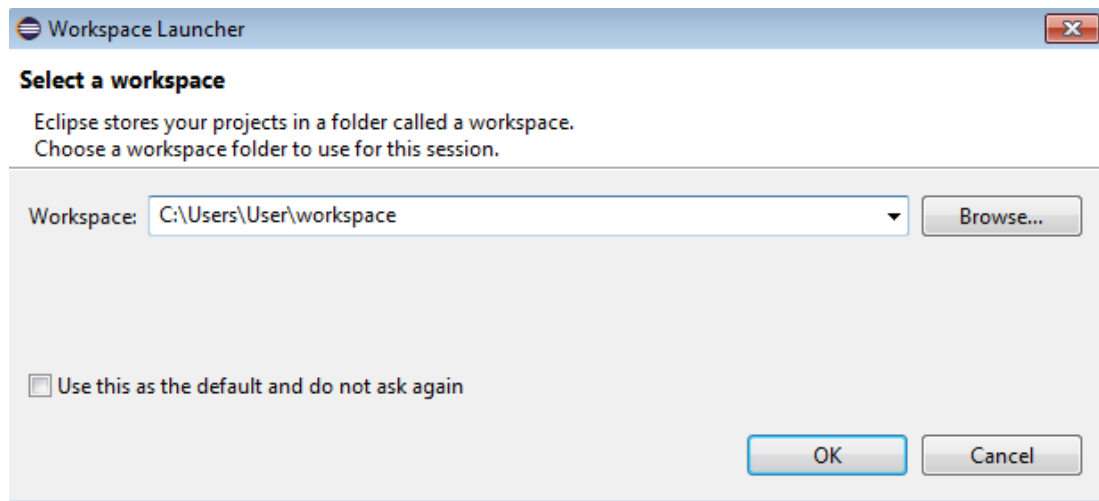


Figure 30: Choose workspace directory.

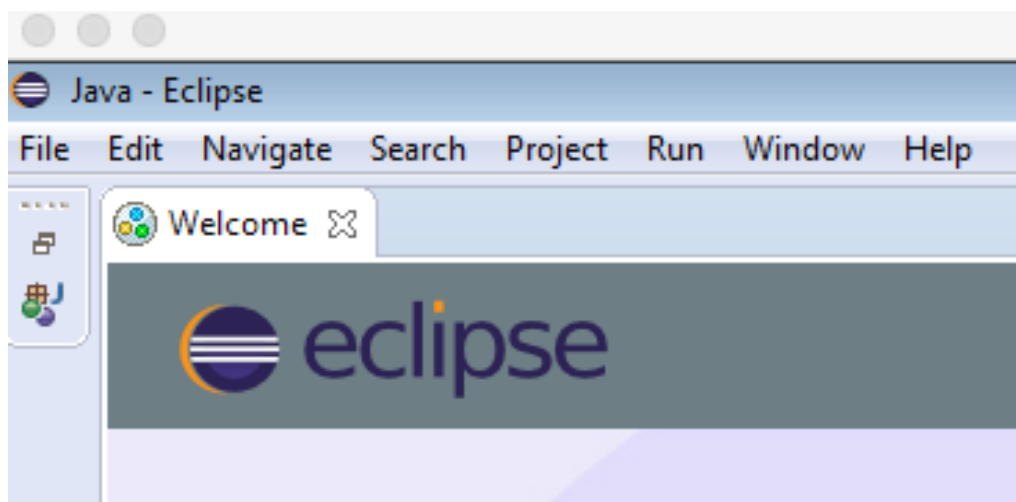


Figure 31: Close the Welcome Screen.

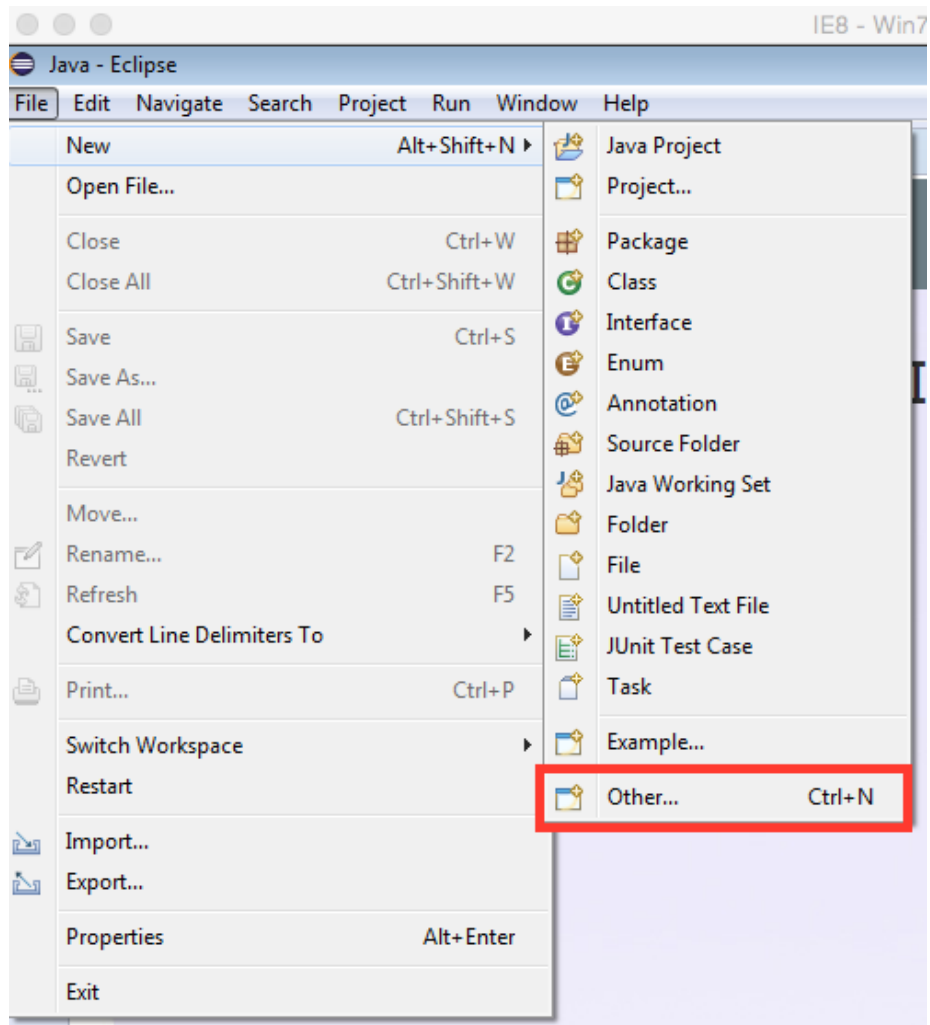


Figure 32: Click on File > New > Other.



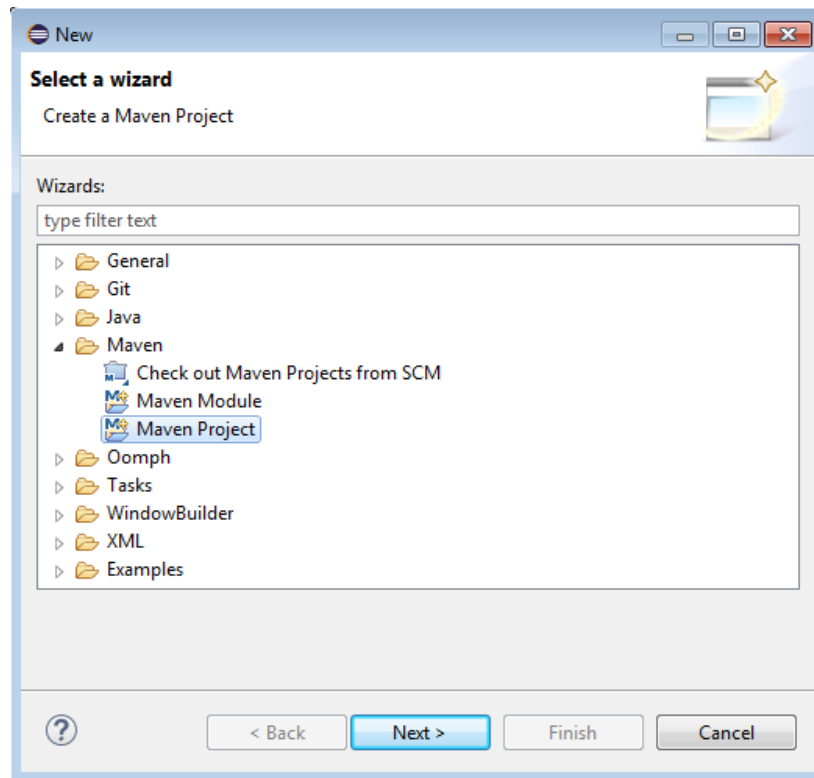


Figure 33: Choose Maven Project.

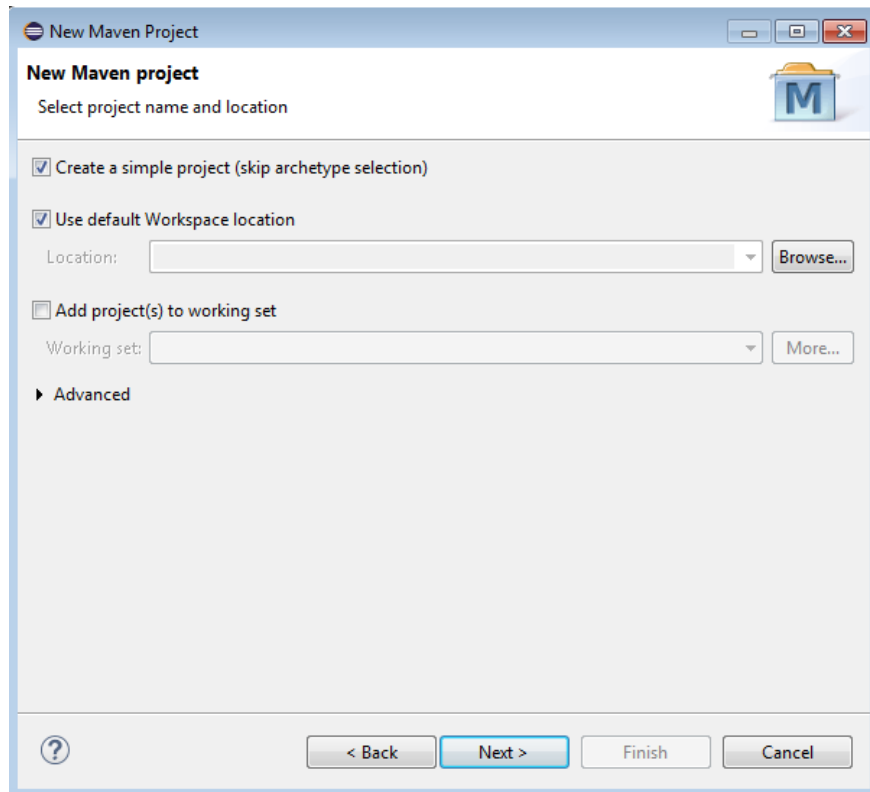


Figure 34: Check Create a Simple Project.

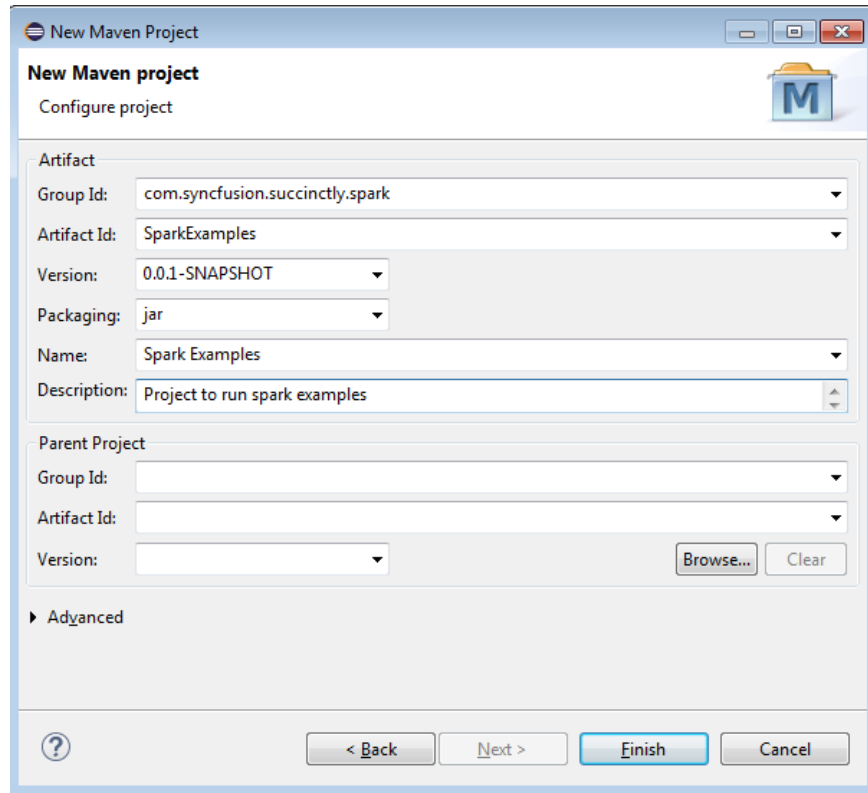


Figure 35: Fill Out New Project Configuration.

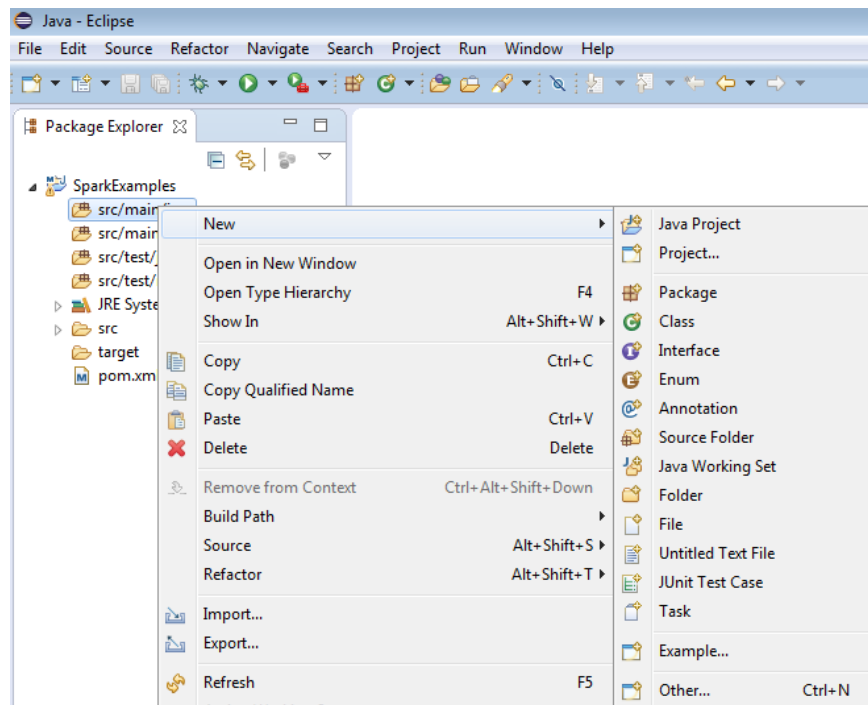


Figure 36: In src/main/java Create New Package.

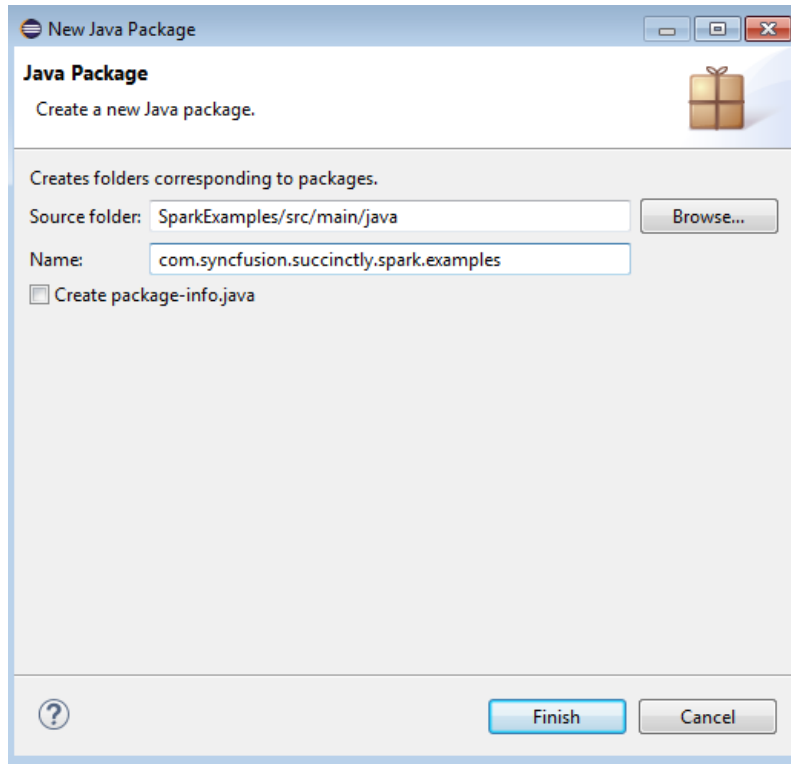


Figure 37: Create new package *com.syncfusion.succinctly.spark.examples*.

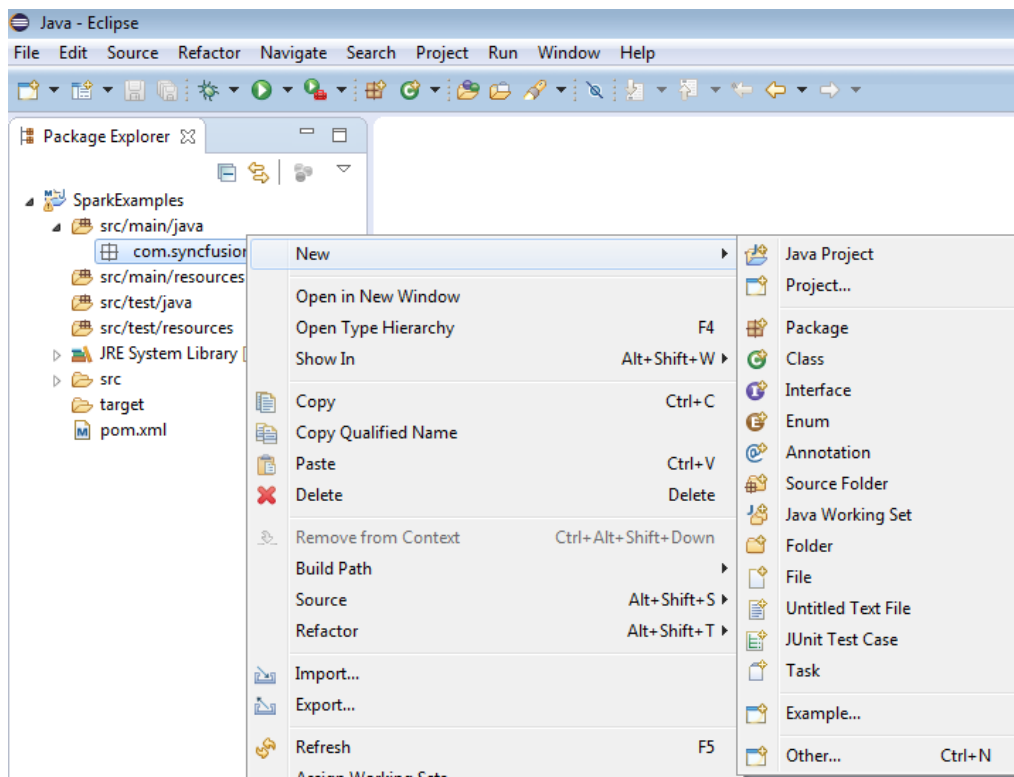


Figure 38: Create New Java Class in created package.

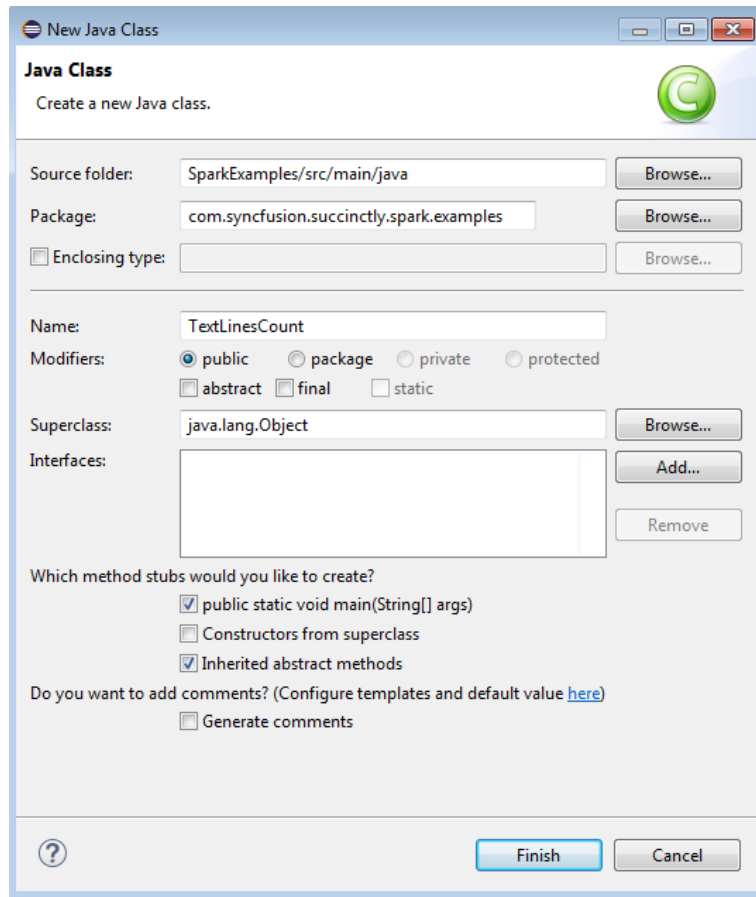


Figure 39: Enter Class Name into New Java Class Dialog, check Create Public Static Void Main.

Enter following code into newly created Java class:

Code Listing 22: Java TextLinesCount class

```
package com.syncfusion.succinctly.spark.examples;

import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.io.UnsupportedEncodingException;

import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.api.java.function.Function;

public class TextLinesCount {

    public static void main(String[] args) throws FileNotFoundException,
    UnsupportedEncodingException {
        if (args.length < 1) {
```

```

        System.err.println("Please provide a full path to the input files");
        System.exit(0);
    }

    if (args.length < 2) {
        System.err.println("Please provide a full path to the output file");
        System.exit(0);
    }

    SparkConf conf = new SparkConf().setAppName("TextLinesCount").setMaster("local");
    JavaSparkContext context = new JavaSparkContext(conf);

    JavaRDD<String> inputFile = context.textFile(args[0]);

    Function<String, Boolean> filterLinesWithSpark = new Function<String, Boolean>() {
        public Boolean call(String arg0) throws Exception {
            return arg0 != null && arg0.contains("Spark");
        }
    };

    JavaRDD<String> sparkMentions = inputFile.filter(filterLinesWithSpark);

    PrintWriter writer = new PrintWriter(args[1]);
    writer.println(sparkMentions.count());
    writer.close();
}
}

```

Now it's time to set up the dependencies. To do this, we'll edit the **pom.xml** file. Locate the **pom.xml** file in the Eclipse folder and double click on it. An editor will open, but since it's a graphical one we will select the **pom.xml** option and edit it manually:

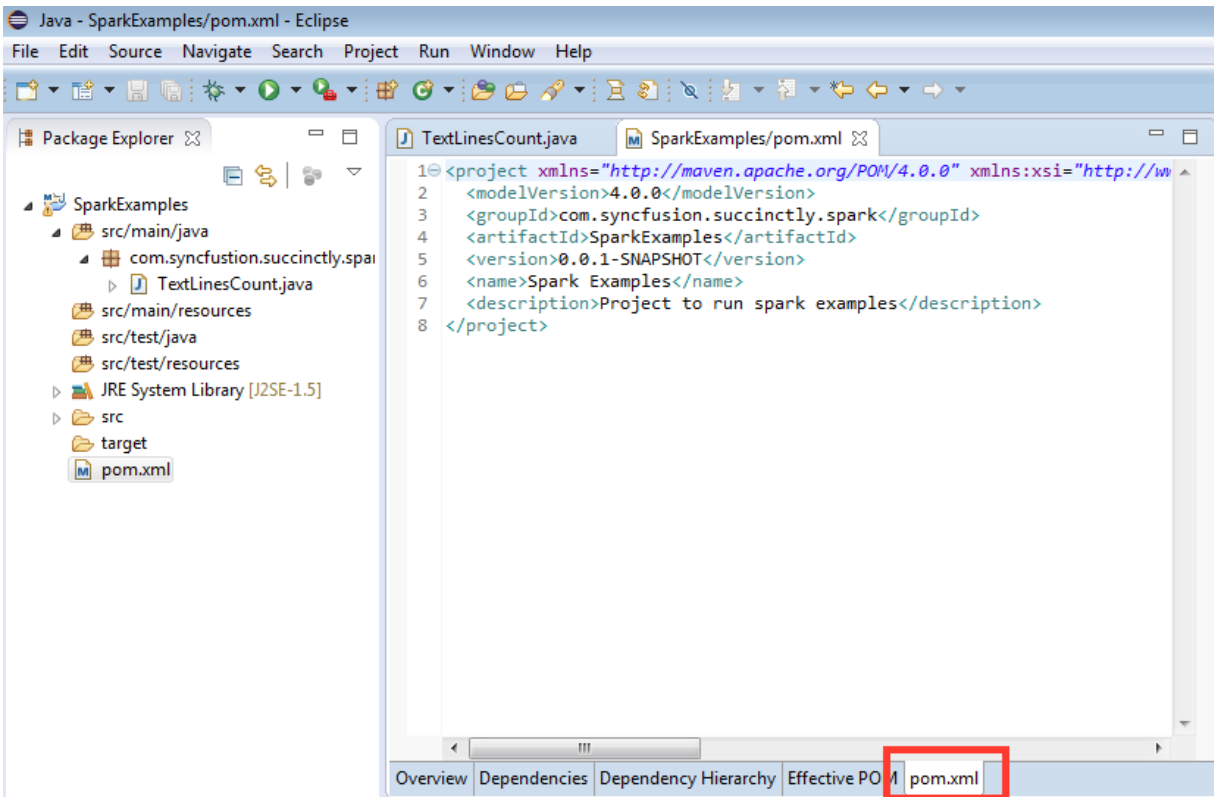


Figure 40: Maven pom.xml File Editor

After opening the editor, change the .xml file to the following:

Code Listing 23: TextLinesCount Maven pom file

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.syncfusion.succinctly.spark</groupId>
  <artifactId>SparkExamples</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>Spark Examples</name>
  <description>Project to run spark examples</description>

  <dependencies>
    <dependency>
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-core_2.10</artifactId>
      <version>1.4.1</version>
    </dependency>
  </dependencies>
</project>
```

```

<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.3</version>
    </plugin>
  </plugins>
</build>

</project>

```

The next step is to package the application:

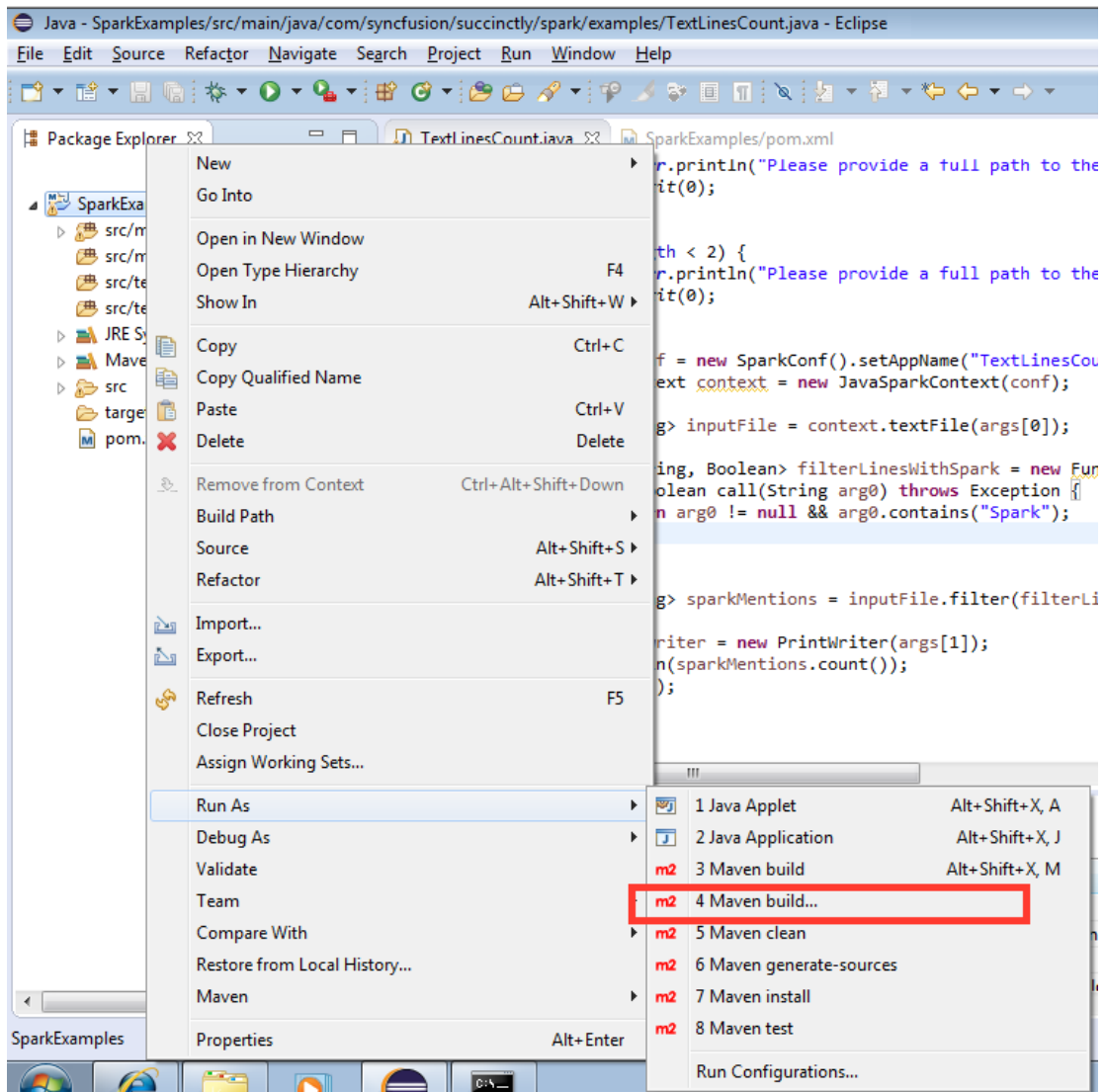


Figure 41: Running Maven Build with Options

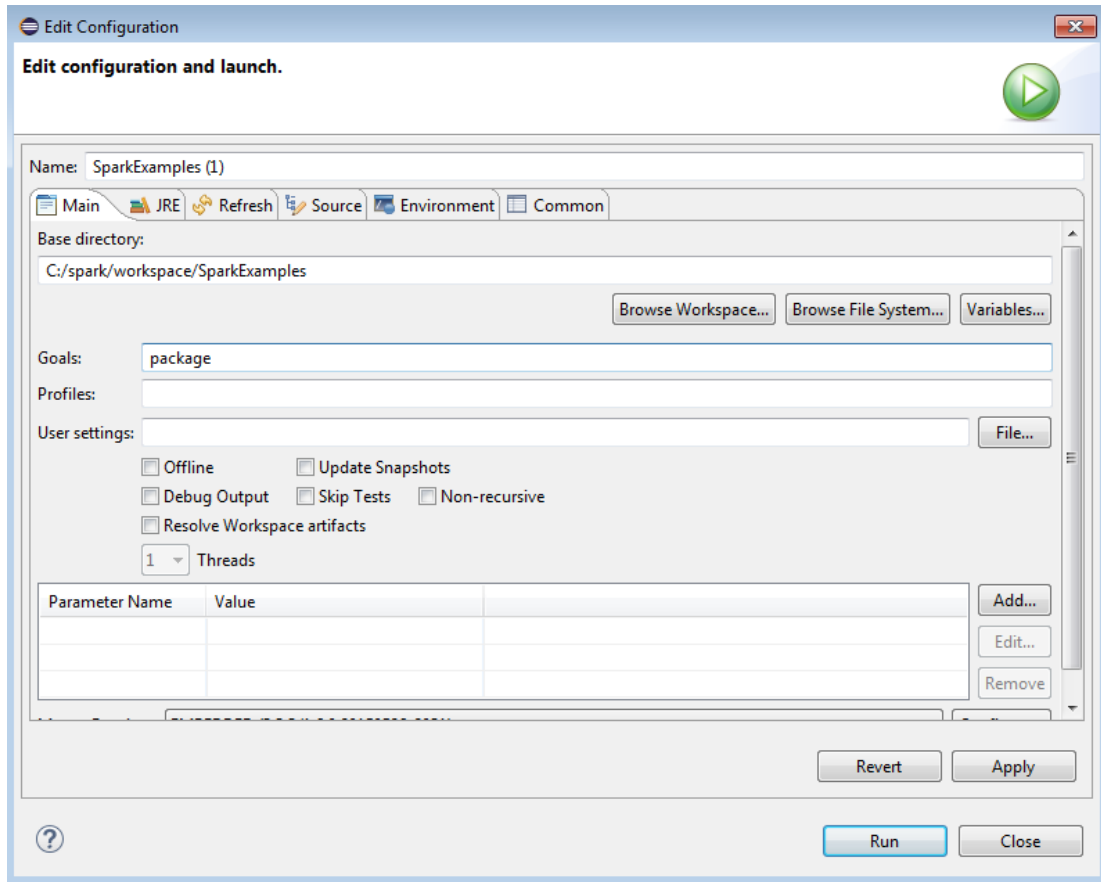


Figure 42: Under Goals type in package and click Run.

The result of the Run operation will be a Console window with the following output. Note that it may vary depending on how you make the setup from previous figures and on the platform that you are using for development:

Code Listing 24: Build Process Result

```
[INFO] Building jar: C:\spark\workspace\SparkExamples\target\SparkExamples-0.0.1-SNAPSHOT.jar
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 4.845 s
[INFO] Finished at: 2015-08-08T13:46:32-07:00
[INFO] Final Memory: 9M/23M
[INFO] -----
```

Remember the location of the .jar file; you are going to need it when submitting a task to Spark. Now it's time to submit the task and give it to Spark for processing. Open the command prompt or the shell and go to the location where you installed Spark. On Windows, the submit command will look something like the following:



Code Listing 25: Task Submit Operation on Windows

```
> bin\spark-submit ^  
--class com.synctfusion.succinctly.spark.examples.TextLinesCount ^  
--master local ^  
file:///C:/spark/workspace/SparkExamples/target/SparkExamples-0.0.1-SNAPSHOT.jar ^  
C:/spark/README.md C:/spark/result.txt
```

If you are running Linux, the submit command will be a bit different:

Code Listing 26: Task submit operation on Linux

```
$ ./bin/spark-submit \  
--class com.synctfusion.succinctly.spark.examples.TextLinesCount \  
--master local \  
/root/spark/SparkExamples-0.0.1-SNAPSHOT.jar \  
/root/spark-1.4.1-bin-hadoop2.6/README.md /root/spark/result.txt
```

After running the submit command a new file, `result.txt`, should be created, and it should contain a number representing total count of lines having the word *Spark* in them. For quite some time now, if you tried to run a previous example on a Windows platform you might run into problems and null pointer exceptions when trying to submit tasks. It is all documented under <https://issues.apache.org/jira/browse/SPARK-2356>. There is a relatively simple solution for this particular problem.



**Note: Spark on Windows will probably have problems while submitting tasks.**

To go around this problem, you need to create a folder somewhere on the disk. In my case, I used `C:\hadoop`. Under it I created a subfolder, `bin`, and I copied a file located under <http://public-repo-1.hortonworks.com/hdp-win-alpha/winutils.exe> into it. I also needed to create an environment variable, `HADOOP_HOME`, and set it to `C:\hadoop` in order to be able to submit a task to Spark and run the example.

We described how to add or change environment variables when we were going through the Java and Scala installation processes, and we won't go over specific steps again. After changing the environment variable, make sure that you restarted your Command Prompt. If environment variable isn't picked up, the submit process will result in failure.

The syntax that we used with this example is Java version 7. As you can see, there are a lot more lines of code than in Scala or Python. This is partially due to the fact that we have to do some initialization of Spark Context and command line parameter processing. We also need to print the results out to a file, but the critical section is defining a function to filter out the lines. We could use Java 8 to keep the number of lines smaller. You have to have JDK version 8 on your computer to be able to write Java 8 programs. Depending on your system configuration, you will probably need to change the `pom.xml` file as shown here:

Code Listing 27: pom.xml File for Java 8 Compatible Project

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.syncfusion.succinctly.spark</groupId>
  <artifactId>SparkExamples</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>Spark Examples</name>
  <description>Project to run spark examples</description>

  <dependencies>
    <dependency>
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-core_2.10</artifactId>
      <version>1.4.1</version>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.3</version>
        <configuration>
          <source>1.8</source>
          <target>1.8</target>
        </configuration>
      </plugin>
    </plugins>
  </build>

</project>
```

The rest of the code will be pretty much the same, but the filtering lines will be much more elegant:

Code Listing 28: Collection Lines Filtered with Java 8 Syntax

```
JavaRDD<String> inputFile = context.textFile(args[0]);
JavaRDD<String> sparkMentions = inputFile.filter(s -> s != null && s.contains("Spark"));

// count the lines having Spark in them
sparkMentions.count();
```

This is actually pretty close to Python and Scala. In this section, we went over how to submit tasks to Spark in Java. We won't go over how to submit Java tasks to Spark until the end of the book; there will only be the code for a Java class, and you can use instructions in this section to submit it to Spark and check it out. Note that the example receives the input and output file as parameters and that you have to provide them when submitting the task. This is due to the fact that Java doesn't support REPL, so we have to do it by printing the results to a file. In the next section we are going to go over a bit more complex examples. We are going to count occurrences of words.

## Counting Word Occurrences

Counting word occurrences is a very common example when it comes to introducing big data processing frameworks or techniques. In this example, we are going to count occurrences of words in a file, and then we are going to order the occurrences by their frequency. In this section, there will be no separate sections for every programming language used. Instead, we are just going to review the examples together. Let's start with Scala:

*Code Listing 29: Scala Word Count in README.md file, Ten Most Common Words*

```
val readmeFile = sc.textFile("README.md")

val counts = readmeFile.flatMap(line => line.split(" ")).
  map(word => (word, 1)).
  reduceByKey(_ + _)

val sortedCounts = counts.sortBy(_._2, false)

sortedCounts.take(10).foreach(println)
```

If everything was fine, you should see the following output:

*Code Listing 30: Top Ten Words in README.md File Produced by Scala*

```
scala> sortedCounts.take(10).foreach(println)
(,66)
(the,21)
(Spark,14)
(to,14)
(for,11)
(and,10)
(a,9)
(##,8)
(run,7)
(can,6)
```

Note that the most common word is actually an empty character, but we are still learning, so we'll just go along with that. Just for this exercise, try to modify the example so that you filter the empty characters out. Here's what the same task would look like using the Python interface:

*Code Listing 31: Python Word Count in README.md file, Ten Most Common Words*

```
readme_file = spark.textFile("README.md")

counts = readme_file.flatMap(lambda line: line.split(" ")) \
    .map(lambda word: (word, 1)) \
    .reduceByKey(lambda a, b: a + b)

counts.takeOrdered(10, key=lambda x: -x[1])
```

The result of running the example in the PySpark shell should be something like:

*Code Listing 32: Top Ten Words in README.md File Produced by Python*

```
>>> counts.takeOrdered(10, key=lambda x: -x[1])
[(u'', 75), (u'the', 21), (u'Spark', 14), (u'to', 14), (u'for', 11), (u'and', 10), (u'a', 9), (u'##', 8), (u'run', 7), (u'is', 6)]
```

Let's have a look at what word count would look like in Java 7.

*Code Listing 33: Top Ten Words in README.md File Produced by Java*

```
package com.syncfusion.succinctly.spark.examples;

import java.util.Arrays;

import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.api.java.function.FlatMapFunction;
import org.apache.spark.api.java.function.Function2;
import org.apache.spark.api.java.function.PairFunction;

import scala.Tuple2;

public class WordCountJava7 {

    public static void main(String[] args) {
        if (args.length < 1) {
            System.err.println("Please provide a full path to the input files");
            System.exit(0);
        }
    }
}
```

```

    if (args.length < 2) {
        System.err.println("Please provide a full path to the output file");
        System.exit(0);
    }

    SparkConf conf = new SparkConf().setAppName("WordCountJava7").setMaster("local");
    JavaSparkContext context = new JavaSparkContext(conf);

    JavaRDD<String> inputFile = context.textFile(args[0]);

    JavaRDD<String> words = inputFile.flatMap(new FlatMapFunction<String, String>() {
        public Iterable<String> call(String s) {
            return Arrays.asList(s.split(" "));
        }
    });

    JavaPairRDD<String, Integer> pairs = words.mapToPair(new PairFunction<String, String,
Integer>() {
        public Tuple2<String, Integer> call(String s) {
            return new Tuple2<String, Integer>(s, 1);
        }
    });

    JavaPairRDD<String, Integer> counts = pairs.reduceByKey(new Function2<Integer, Integer,
Integer>() {
        public Integer call(Integer a, Integer b) {
            return a + b;
        }
    });

    JavaPairRDD<Integer, String> swappedPairs = counts
        .mapToPair(new PairFunction<Tuple2<String, Integer>, Integer, String>() {
            public Tuple2<Integer, String> call(Tuple2<String, Integer> item) throws Exception {
                return item.swap();
            }
        });

    JavaPairRDD<Integer, String> wordsSorted = swappedPairs.sortByKey(false);

    wordsSorted.saveAsTextFile(args[1]);
}
}

```

In the final step, we are saving the collection directly to the disk by using the Spark-provided function `saveAsTextFile`. If you submit a task with a previously defined Java class to Spark, you might run into `FileAlreadyExistsException` if you run it with the same output parameter as on the previous Java examples. If that happens, simply delete that file. The output might confuse you a bit; it will be a directory. If you submitted the task with the same parameters as in earlier examples, you might find the output to be something like this:

*Code Listing 34: Java 7 Example Output Directory Structure*

```
.
├─ results.txt
│   └─ _SUCCESS
│       └─ part-00000
```

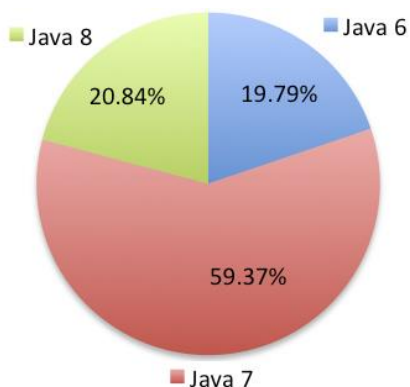
1 directory, 2 files

*Code Listing 35: Java Example Output File Content*

```
(66,)
(21,the)
(14,Spark)
(14,to)
(11,for)
(10,and)
(9,a)
(8,##)
(7,run)
(6,is)
```

...

Again, the reason why there are examples for Java 7 and Java 8 is that, at the moment, Java 7 is still the most used one while Java 8 provides much less code and is gaining a lot of momentum. Here is the situation with market share in 2015:



*Figure 43: Java Versions Market Shares in 2015*

Because of that, I'll also include examples for Java 8. Here is the word count example written in Java 8. When it comes to the lines that do the work, there's a lot less code involved:

*Code Listing 36: Java 8 Word Count Example*

```
package com.syncfusion.succinctly.spark.examples;

import java.util.Arrays;

import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;

import scala.Tuple2;

public class WordCountJava8 {

    public static void main(String[] args) {
        if (args.length < 1) {
            System.err.println("Please provide a full path to the input files");
            System.exit(0);
        }

        if (args.length < 2) {
            System.err.println("Please provide a full path to the output file");
            System.exit(0);
        }

        SparkConf conf = new SparkConf().setAppName("WordCountJava7").setMaster("local");
        JavaSparkContext context = new JavaSparkContext(conf);

        JavaRDD<String> inputFile = context.textFile(args[0]);

        JavaPairRDD<String, Integer> counts = inputFile.flatMap(line -> Arrays.asList(line.split(" ")))
            .mapToPair(w -> new Tuple2<String, Integer>(w, 1))
            .reduceByKey((x, y) -> x + y);

        JavaPairRDD<Integer, String> wordsSorted = counts.mapToPair(item -> item.swap())
            .sortByKey(false);

        wordsSorted.saveAsTextFile(args[1]);
    }
}
```

The output should be the same as in Java 7. This completes the basic examples. In the next chapter we are going to dive deep into mechanisms provided by Spark.

# Chapter 3 Spark Internals

In previous chapters, the orientation was more towards practicality and how to get things going with Spark. In this chapter, we are going to go into the concepts behind Spark. We are taking a step back because you will be able to harness the power of Spark in a much better fashion if you understand where all the speed is coming from. In Java source, you might have noticed classes that had RDD in their name. RDD stands for *resilient distributed dataset*. We mentioned them earlier but didn't go into them too much. They are often described as being simple collections. In practice, you can treat them that way, and it's perfectly fine, but there is much more behind them. Let's dive into the concept of RDD.

## Resilient Distributed Dataset (RDD)

We've mentioned the concept of RDD but haven't gone much deeper into it. Let's have a look at what an RDD really is. An RDD is an immutable and distributed collection of elements that can be operated upon in parallel. It is the basic abstraction that Spark uses to function. RDDs support various kinds of transformations on them. We already used some of them, like **map** and **filter**, in previous examples. Every transformation creates a new RDD. One important thing to remember about RDDs is that they are lazily evaluated; when transformation is called upon them, no actual work is done right away. Only the information about the source of RDD is stored and the transformation has to be applied.



**Note:** *RDDs are lazily evaluated data structures. In short, that means there is no processing associated with calling transformations on RDDs right away.*

This confuses many developers at first because they perform all the transformations and at the end don't get the results that they expect. That's totally fine and is actually the way Spark works. Now you may be wondering: how do we get results? The answer to this is by calling actions. Actions trigger the computation operations on RDDs. In our previous examples, actions were called by using **take** or **saveAsTextFile**.



**Note:** *Calling actions trigger computation operations. Action = Computation*



There are a lot of benefits from the fact that the transformations are lazily evaluated. Some of them are that operations can be grouped together, reducing the networking between the nodes processing the data; there are no multiple passes over the same data. But there is a pitfall associated with all this. Upon user request for action, Spark will perform the calculations. If the data between the steps is not cached, Spark will reevaluate the expressions again because RDDs are not materialized. We can instruct Spark to materialize the computed operations by calling **cache** or **persist**. We will talk about the details soon. Before we go into how caching works, it's important to mention that the relations between the RDDs are represented as a directed acyclic graph, or DAG. Let's break it down by looking into a piece of Scala code:

*Code Listing 37: Scala Word Count in README.md file, Ten Most Common Words*

```
val readmeFile = sc.textFile("README.md")

val counts = readmeFile.flatMap(line => line.split(" ")).
  map(word => (word, 1)).
  reduceByKey(_ + _)

val sortedCounts = counts.sortBy(_._2, false)

sortedCounts.take(10).foreach(println)
```

An RDD is created from a **README.md** file. Calling a **flatMap** transformation on the lines in the file creates an RDD too, and then a pair RDD is created with a word and a number one. Numbers paired with the same word are then added together. After that, we sort them by count, take the first ten, and print them. The first action that triggers computation is **sortBy**. It's a Scala-specific operation, and it needs concrete data to do the sort so that computation is triggered. Let's have a look at the DAG up until that point:

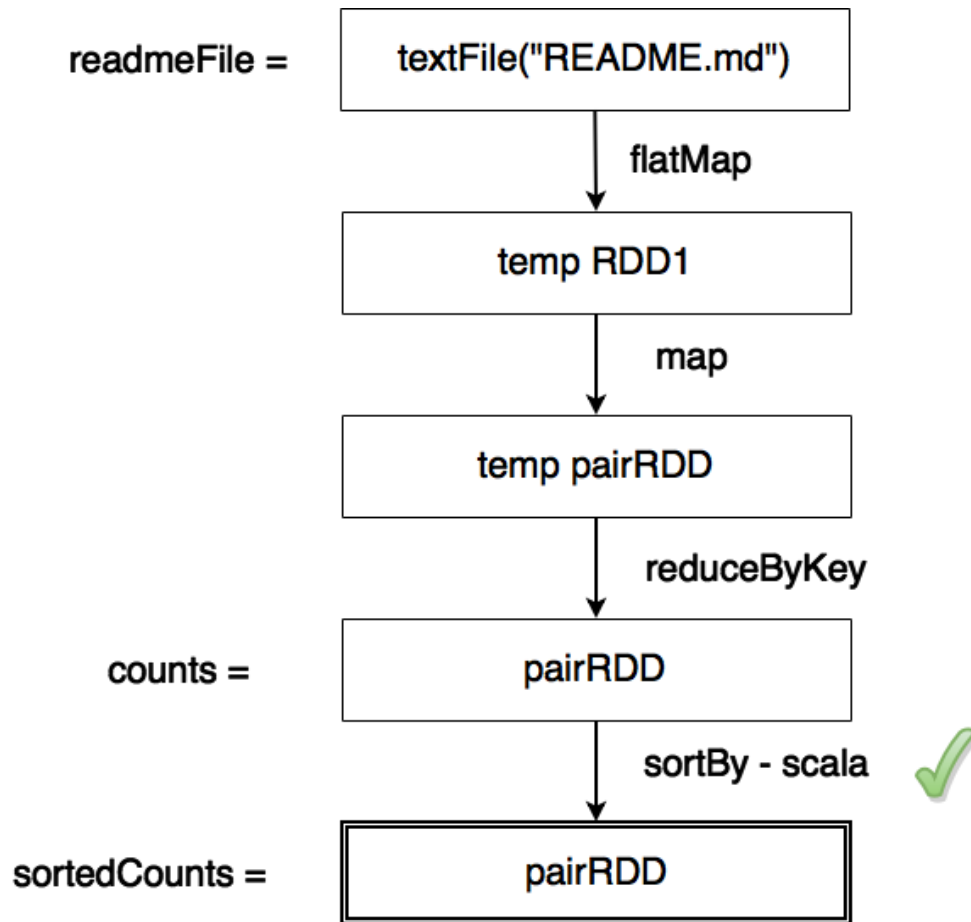


Figure 44: Compute Operation Directed Acyclic Graph

Figure 44 shows how RDDs are computed. Vertices are RDDs and edges are transformations and actions on RDDs. Only actions can trigger computation. In our example, this happens when we call the **sortBy** action. There are a fair number of transformations in Spark, so we are only going to go over the most common ones.

Table 3: Common Spark Transformations

Transformation	Description
<code>filter(func)</code>	Return a new dataset formed by selecting those elements of the source on which <i>function</i> returns true.
<code>map(func)</code>	Return a new distributed dataset formed by passing each element of the source through a function.
<code>flatMap(func)</code>	Similar to map, but each input item can be mapped to 0 or more output items (so the function should return a sequence rather than a single item).

Transformation	Description
<code>distinct([numTasks])</code>	Return a new dataset that contains the distinct elements of the source dataset.
<code>sample(withReplacement, fraction, seed)</code>	Sample a fraction of the data, with or without replacement, using a given random number generator seed. Often used when developing or debugging.
<code>union(otherDataset)</code>	Return a new dataset that contains the union of the elements in the source dataset and the argument.
<code>intersection(otherDataset)</code>	Return a new RDD that contains the intersection of elements in the source dataset and the argument.
<code>cartesian(otherDataset)</code>	When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements).
Source: <a href="http://spark.apache.org/docs/latest/programming-guide.html#transformations">http://spark.apache.org/docs/latest/programming-guide.html#transformations</a>	

Most of the people starting out with Spark get confused with **flatMap** and **map** functions. The easiest way to wrap your head around it is to remember that **map** is a one-to-one function and **flatMap** is one to many. Input for **flatMap** is usually a collection of elements. In our examples, we started out with lines of text. The lines of text were then split into words. Every line contains zero or multiple words. But we want every word to appear in the final RDD separately, so we used a **flatMap** function. Just so that it stays with you longer, let's have a look at the following figure:

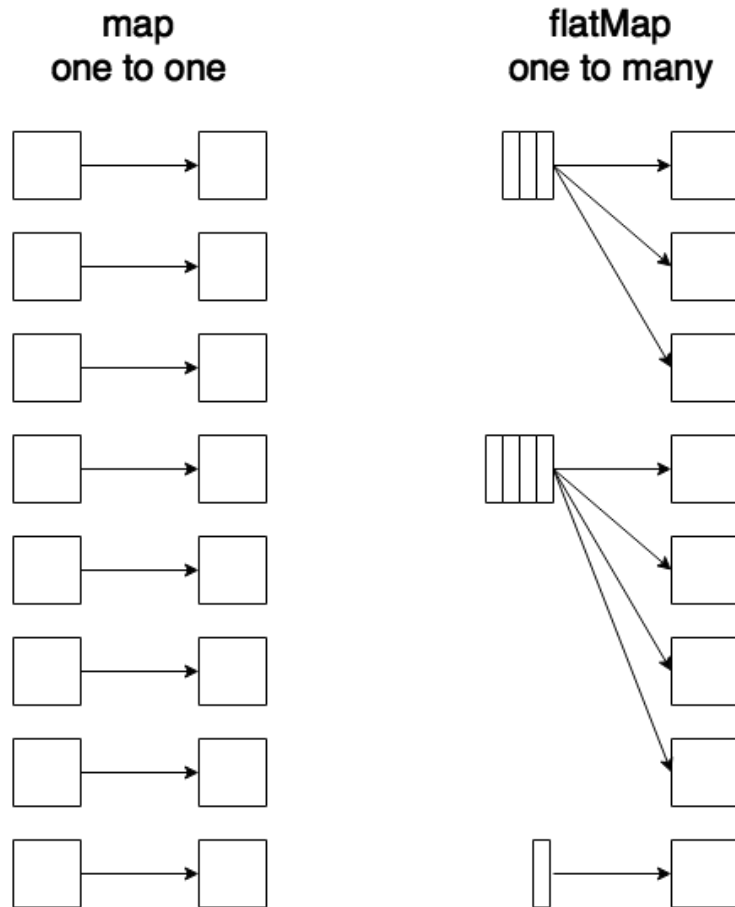


Figure 45: Spark Transformations Map and flatMap in Action

Some of the mentioned transformations are mostly used with simple RDDs, while some are more often used with pair RDDs. We'll just mention them for now and will go in depth later. Let's look at some of the most common actions:

Table 4: Common Spark Actions

Transformation	Description
<code>collect()</code>	Return all the elements of a dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of data. Be careful when calling this.
<code>count()</code>	Return the number of elements in the dataset.
<code>first()</code>	Return the first element of the dataset (similar to <code>take(1)</code> ).

Transformation	Description
take(n)	Return an array with the first $n$ elements of the dataset.
takeSample(withReplacement, num, [seed])	Return an array with a random sample of “num” elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.
reduce(func)	Aggregate the elements of dataset using a function (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
foreach(func)	Run a function on each element of the dataset. Usually used to interact with external storage systems.
saveAsTextFile(path)	Write the elements of the dataset as a text file (or set of text files) in a given directory in the local file system, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file.
Source: <a href="http://spark.apache.org/docs/latest/programming-guide.html#actions">http://spark.apache.org/docs/latest/programming-guide.html#actions</a>	

Now that we’ve covered most commonly used actions and transformations, it might be a good thing to mention some helper operations that come in very handy when developing applications with Spark. One of them is `toDebugString`:

*Code Listing 38: Result of Running toDebugString in Spark Shell*

```
scala> sortedCounts.toDebugString

res6: String =
(2) MapPartitionsRDD[26] at sortBy at <console>:25 []
| ShuffledRDD[25] at sortBy at <console>:25 []
+- (2) MapPartitionsRDD[22] at sortBy at <console>:25 []
| ShuffledRDD[21] at reduceByKey at <console>:25 []
+- (2) MapPartitionsRDD[20] at map at <console>:24 []
| MapPartitionsRDD[19] at flatMap at <console>:23 []
| MapPartitionsRDD[18] at textFile at <console>:21 []
| README.md HadoopRDD[17] at textFile at <console>:21 []
```

The result is a text representation of the Directed Acyclic Graph associated with the value or variable upon which we called a **toDebugString** method. Earlier we mentioned that we could avoid unnecessary computation by caching results. We'll cover this in the next section.

## Caching RDDs

Most of the time the directed acyclic graph will not be simple hierarchical structure. In practice it may involve a lot of levels that branch into new levels.

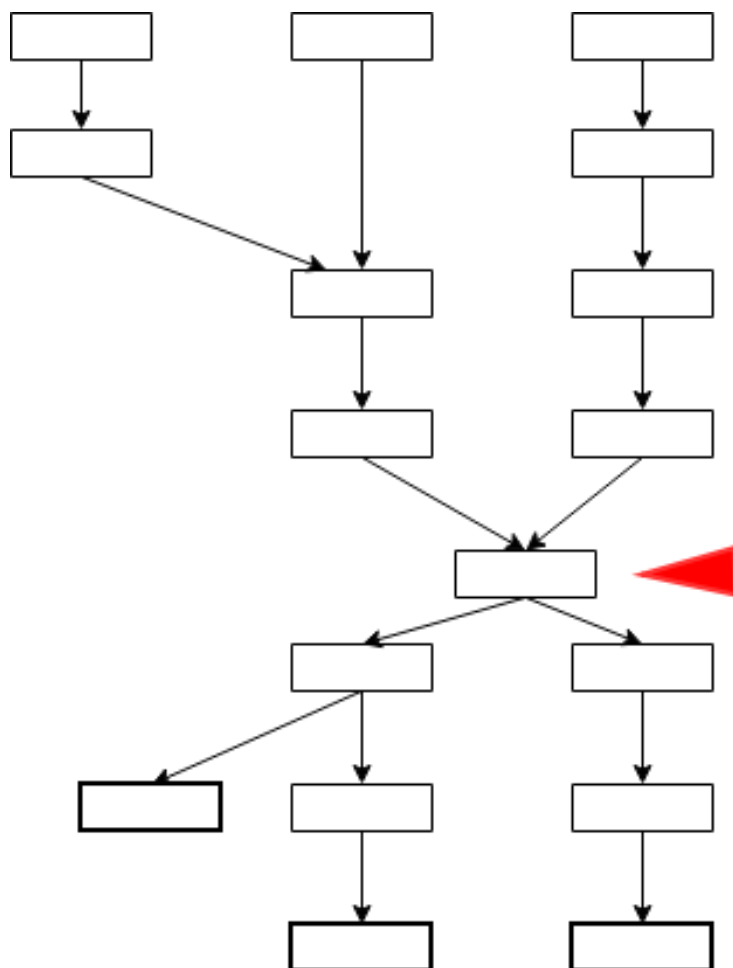


Figure 46: Complex Directed Acyclic Graph with Caching Candidate

When we call actions at the end for every called action because of the lazy evaluation, the computation goes all the way to the beginning for every one of them. It's a good practice to cache RDDs that are sources for generating a lot of other RDDs. In Figure 46, the RDD marked with a red marker is, definitely, a candidate for caching. There are two methods that you can use for caching the RDD. One is **cache** and the other is **persist**. Method **cache** is just a shorthand for **persist**, where a default storage level **MEMORY\_ONLY** is used. Let's have a look at cache storage levels:

Table 5: Spark Storage Levels

Storage Level	Description
MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER	Store RDD as serialized Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
MEMORY_AND_DISK_SER	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	Same as previous levels, but replicate each partition on two cluster nodes.
OFF_HEAP	Is in experimental phase.
Source: <a href="http://spark.apache.org/docs/latest/programming-guide.html#rdd-persistence">http://spark.apache.org/docs/latest/programming-guide.html#rdd-persistence</a>	

Spark systems are usually oriented towards having a lot of CPU cores and a lot of RAM. You have to have very good reasons to store something on disk. Most of the time this option is only used with SSD disks. There was a comparison table earlier in the book when we talked about basics of big data processing where a CPU cycle was a baseline representing one second. RAM access was six minutes and access to SSD disk two to six days. Rotational disk, by comparison, takes one to twelve months. Remember that fact when persisting your data to the disk.



**Tip: Spark needs cores and RAM; most of the time it's advisable to avoid disk.**

This doesn't mean that you should avoid it at all cost, but you have to be aware of significant performance drops when it comes to storing data on disk. If you remember the story about Spark's sorting record, the amount of data sorted definitely didn't fit only into RAM, and disk was also heavily used. It's just important to be aware of the fact that disk is significantly slower than RAM. Sometimes even going to disk might be a much better option than fetching elements from various network storages or something similar. Caching definitely comes in handy in those situations. Let's have a look how we would cache an RDD. In this chapter, we are mostly using Scala language because we are discussing concepts and not implementations in specific languages. Let's cache an RDD:

*Code Listing 39: Calling Cache on RDD*

```
scala> sortedCounts.cache()
res7: sortedCounts.type = MapPartitionsRDD[26] at sortBy at <console>:25

scala> sortedCounts.take(10).foreach(println)
(,66)
(the,21)
(Spark,14)
(to,14)
(for,11)
(and,10)
(a,9)
(##,8)
(run,7)
(can,6)
```

RDDs are evaluated lazily, so just calling cache has no effect. That's why we are taking the first ten elements from the RDD and printing them. We are calling action and calling action triggers caching to take place.

We didn't mention it earlier, but when you run a Spark or Python shell it also starts a small web application that gives you overview over resources the shell uses, environment, and so on. You can have a look at it by opening your browser and navigating to <http://localhost:4040/>. The result of this action should look something like:



Spark 1.4.1		Jobs	Stages	Storage	Environment	Executors	Spark shell application UI	
-------------	--	------	--------	---------	-------------	-----------	----------------------------	--

**Spark Jobs (?)**

Total Uptime: 10.6 h  
Scheduling Mode: FIFO  
Completed Jobs: 8  
[Event Timeline](#)

**Completed Jobs (8)**

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
7	<a href="#">take at &lt;console&gt;:28</a>	2015/08/09 20:26:45	0.1 s	2/2 (1 skipped)	3/3 (2 skipped)
6	<a href="#">sortBy at &lt;console&gt;:25</a>	2015/08/09 16:38:36	84 ms	2/2	4/4
5	<a href="#">take at &lt;console&gt;:28</a>	2015/08/09 10:08:04	33 ms	2/2 (1 skipped)	3/3 (2 skipped)
4	<a href="#">sortBy at &lt;console&gt;:25</a>	2015/08/09 10:07:56	27 ms	1/1 (1 skipped)	2/2 (2 skipped)
3	<a href="#">take at &lt;console&gt;:28</a>	2015/08/09 10:07:28	0.2 s	2/2 (1 skipped)	3/3 (2 skipped)
2	<a href="#">sortBy at &lt;console&gt;:25</a>	2015/08/09 10:07:09	49 ms	1/1 (1 skipped)	2/2 (2 skipped)
1	<a href="#">take at &lt;console&gt;:26</a>	2015/08/09 10:05:27	0.4 s	2/2	3/3
0	<a href="#">take at &lt;console&gt;:24</a>	2015/08/09 10:02:19	0.2 s	1/1	1/1

Figure 47: Spark Shell Application UI

By default, Spark Shell Application UI opens the Jobs tab. Navigate to the Storage tab:

Spark 1.4.1		Jobs	Stages	Storage	Environment	Executors	Spark shell application UI	
-------------	--	------	--------	---------	-------------	-----------	----------------------------	--

**Storage**

RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size in ExternalBlockStore	Size on Disk
<a href="#">MapPartitionsRDD</a>	Memory Deserialized 1x Replicated	1	50%	6.3 KB	0.0 B	0.0 B

Figure 48: Spark Shell Application UI Storage Details

The RDD that we cached by calling `cache` is visible in the Spark Shell Application UI Storage Tab. We can even find out details by clicking on RDD Name:

Spark 1.4.1		Jobs	Stages	Storage	Environment	Executors	Spark shell application UI	
-------------	--	------	--------	---------	-------------	-----------	----------------------------	--

**RDD Storage Info for MapPartitionsRDD**

Storage Level: Memory Deserialized 1x Replicated  
Cached Partitions: 1  
Total Partitions: 2  
Memory Size: 6.3 KB  
Disk Size: 0.0 B

**Data Distribution on 1 Executors**

Host	Memory Usage	Disk Usage
localhost:50889	6.3 KB (265.1 MB Remaining)	0.0 B

**1 Partitions**

Block Name	Storage Level	Size in Memory	Size on Disk	Executors
rdd_26_0	Memory Deserialized 1x Replicated	6.3 KB	0.0 B	localhost:50889

Figure 49: Spark Shell Application UI Storage Details for a Single RDD

If you want to stop caching your RDDs in order to free up resources for further processing, you can simply call the **unpersist** method on a particular RDD. This method has an immediate effect, and you don't have to wait for when the RDD is recomputed. Here is an example on how **unpersist** works:

*Code Listing 40: Calling Unpersist on RDD*

```
scala> sortedCounts.unpersist()

res3: sortedCounts.type = MapPartitionsRDD[9] at sortBy at <console>:25
```

You can check the effects of calling **cache** and **unpersist** methods in Spark's WebUI, which we showed in earlier figures.

By now we covered enough to get you going with RDDs. In previous examples, we just mentioned Pair RDDs and worked with them a bit, but we didn't discuss them in depth. In the next section we are going to explain how to use Pair RDDs.

## Pair RDDs

Pair RDDs behave pretty similar to basic RDDs. The main difference is that RDD elements are key value pairs, and key value pairs are a natural fit for distributed computing problems. Pair RDDs are heavily used to perform various kinds of aggregations and initial Extract Transform Load procedure steps. Performance-wise there's one important, rarely mentioned fact about them: Pair RDDs don't spill on disk. Only basic RDDs can spill on disk. This means that a single Pair RDD must fit into computer memory. If Pair RDD content is larger than the size of the smallest amount of RAM in the cluster, it can't be processed. In our previous examples we already worked with Pair RDDs, we just didn't go in depth with them.

We'll start with the simplest possible example here. We are going to create RDDs from a collection of in-memory elements defined by ourselves. To do that, we will call the **parallelize** Spark context function. Java API is a bit more strongly typed and can't do automatic conversions, so Java code will be a bit more robust since we have to treat the elements with the right types from the start. In this example, there are no significant elegance points gained for using Java 8, so we'll stick with Java 7. As in the previous chapters let's start with Scala:

*Code Listing 41: Simple Pair RDD Definition in Scala*

```
val pairs = List((1,1), (1,2), (2,3), (2, 4), (3, 0))

val pairsRDD = sc.parallelize(pairs)

pairsRDD.take(5)
```

*Code Listing 42: Simple Pair RDD Definition in Scala Result*

```
scala> pairsRDD.take(5)
```

```
res2: Array[(Int, Int)] = Array((1,1), (1,2), (2,3), (2,4), (3,0))
```

*Code Listing 43: Simple Pair RDD Definition in Python*

```
pairs = [(1, 1), (1, 2), (2, 3), (2, 4), (3, 0)]

pairs_rdd = sc.parallelize(pairs)

pairs_rdd.take(5)
```

*Code Listing 44: Simple Pair RDD Definition in Python Result*

```
>>> pairs_rdd.take(5)
[(1, 1), (1, 2), (2, 3), (2, 4), (3, 0)]
```

As mentioned earlier, the Java example will include a few more details. There is no interactive shell for Java, so we are simply going to output the results of Java pair RDD creation to a text file. To define pairs we are going to use the `Tuple2` class provided by Scala. In the previous example, the conversion between the types is done automatically by Scala, so it's a lot less code than with Java. Another important difference is that in Java **parallelize** works only with basic RDDs. To create pair RDDs in Java, we have to use the **parallelizePairs** call from initialized context. Let's start by defining a class:

*Code Listing 45: Simple Pair RDD Definition in Java*

```
package com.synCFusion.succinctly.spark.examples;

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

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

import scala.Tuple2;

public class PairRDDJava7 {
    public static void main(String[] args) {
        if (args.length < 1) {
            System.err.println("Please provide a full path to the output file");
            System.exit(0);
        }

        SparkConf conf = new SparkConf().setAppName("PairRDDJava7").setMaster("local");
        JavaSparkContext context = new JavaSparkContext(conf);

        List<Tuple2<Integer, Integer>> pairs = Arrays.asList(
```

```

        new Tuple2(1, 1), new Tuple2<Integer, Integer>(1, 2),
        new Tuple2<Integer, Integer>(2, 3), new Tuple2<Integer, Integer>(2, 4),
        new Tuple2<Integer, Integer>(3, 0));

    JavaPairRDD<Integer, Integer> pairsRDD = context.parallelizePairs(pairs);

    pairsRDD.saveAsTextFile(args[0]);
}
}

```

Just as a short refresher, to submit the class to Spark you will need to rerun the packaging steps with Maven that are described in earlier chapters. Note that the way you submit may vary depending on what operating system you are using:

*Code Listing 46: Submitting PairRDDJava7 to Spark on Linux Machines*

```

$ ./bin/spark-submit \
--class com.syncfusion.succinctly.spark.examples.PairRDDJava7 \
--master local \
/root/spark/SparkExamples-0.0.1-SNAPSHOT.jar \
/root/spark/resultspair

```

When jobs are submitted, the results of operations are usually not outputted in a log. We will show additional ways to store data in permanent storage later. For now, the results will be saved to a folder. Here is the resulting output file content:

*Code Listing 47: Output File for PairRDDJava7 Task Submitted to Spark*

```

(1,1)
(1,2)
(2,3)
(2,4)
(3,0)

```

Previous examples show how to define Pair RDDs from basic in-memory data structures. Now we are going to go over basic operations with Pair RDDs. Pair RDDs have all the operations that the basic ones have, so, for instance, you can filter them out based on some criteria.

## Aggregating Pair RDDs

This is a very common use case when it comes to Pair RDDs. When we did a word count, we used the **reduceByKey** transformation on a Pair RDD. When **reduceByKey** is called, a pair RDD is returned with the same keys as the original RDD, and the values are aggregation done on every value having the same key. Let's have a look at what the result would be for calling it on pairs (1, 1), (1, 2), (2, 3), (2, 4), and (3, 0), which we used in previous examples. We are going to take five elements, although we know the result won't necessarily have a size of all the elements combined. This is just to trigger computing and to display some results. Again, to run with Java you will have to submit a jar to Spark. To keep things shorter, in Java we will just show the functional part of code in the examples:

*Code Listing 48: reduceByKey in Action with Scala*

```
scala> pairsRDD.reduceByKey(_+_).take(5)
res4: Array[(Int, Int)] = Array((1,3), (2,7), (3,0))
```

*Code Listing 49: reduceByKey in Action with Python*

```
>>> pairs_rdd.reduceByKey(lambda a, b: a + b).take(5)
[(1, 3), (2, 7), (3, 0)]
```

*Code Listing 50: reduceByKey in Action with Java 7*

```
JavaPairRDD<Integer, Integer> reduceByKeyResult = pairsRDD
    .reduceByKey(new Function2<Integer, Integer, Integer>() {
        public Integer call(Integer a, Integer b) {
            return a + b;
        }
    });

reduceByKeyResult.saveAsTextFile(args[0]);
```

*Code Listing 51: reduceByKey in Action with Java 8*

```
JavaPairRDD<Integer, Integer> reduceByKeyResult = pairsRDD.reduceByKey((a, b) -> a + b);

reduceByKeyResult.saveAsTextFile(args[0]);
```

Note that with Java we didn't call the take operation because saving RDDs into a text file iterates over all RDD elements anyway. The resulting file will look like the following:

*Code Listing 52: Java reduceByKey Output*

```
(1,3)
(3,0)
(2,7)
```

Let's go a bit further with the concept. The example was relatively simple, because all we had to do was add everything up. It can get quite complicated in real life, though. For example, calculating average value for a key is a pretty common problem in practice. It's not straightforward and there are some steps to it. In the first step, we have to determine the total count of all the key occurrences. The second step is to combine all the elements and add them up by key. In the final step, we have to divide the sum by total counts per key. To achieve this goal we are going to use the **combineByKey** function. It's a bit complex, so we will break it down into parts. To make explaining easier, we are again going to use our list  $((1,1), (1,2), (2,3), (2,4), (3,0))$ . We will display it graphically after giving textual explanation just to make sure you understand it. I know it might feel a bit challenging at first, but try to read this section a couple of times if it doesn't sit right away. Let's start with primary explanation:

1. Create a Combiner – First aggregation step done for every key. In essence, for every key we create a list of pairs containing values and a value 1.
2. Merge a Value – Per every key we are going to add up the sums and increase the total number of elements by one.
3. Merge two Combiners – In the final step we are going to add up all the sums together with all the counts. This is also necessary because parts of the calculation are done on separate machines. When the results come back we combine them together.

We are going to go over examples in programming languages soon. Before giving any code samples, let's have a look at a graphical description, just so that it sits better. To some of you, the final step might look a bit unnecessary, but remember that Spark processes data in parallel and that you have to combine the data that comes back from all of the nodes in the cluster. To make our example functional in the final step (shown in the following figure) we have to go over all of the returned results and divide the sum with count. But that's just basic mapping, and we won't go much into it. Here is a figure showing how **combineByKey** works:

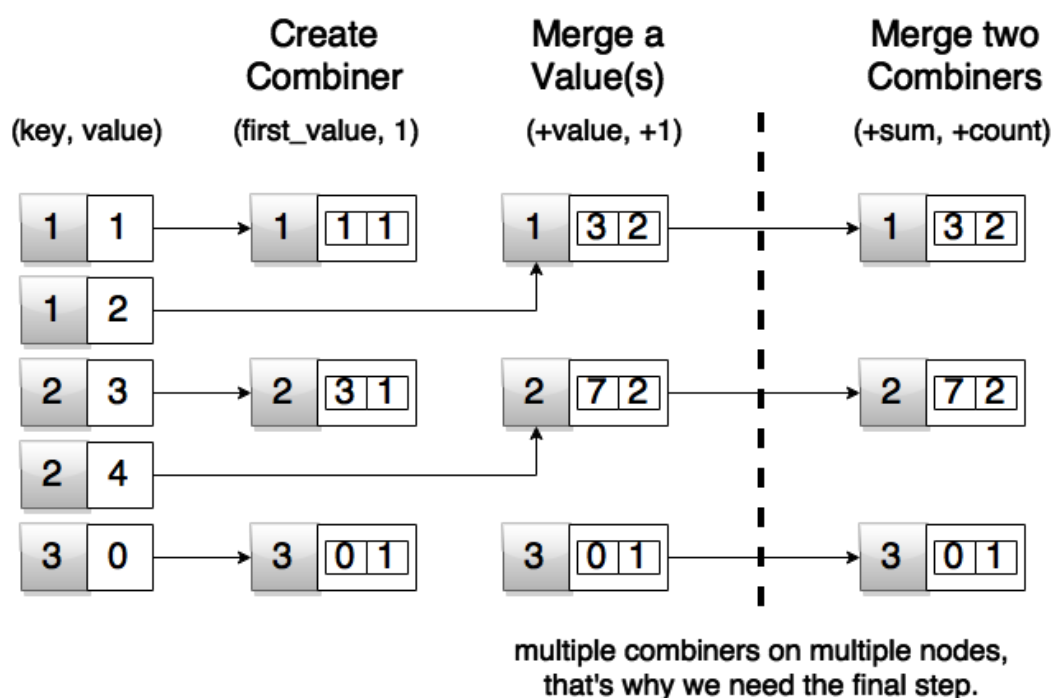


Figure 50: Spark **combineByKey** in Action

Here is what the code would look like in Scala:

*Code Listing 53: Spark combineByKey Scala Code*

```
val pairAverageResult = pairsRDD.combineByKey(
  (v) => (v, 1),
  (acc: (Int, Int), v) => (acc._1 + v, acc._2 + 1),
  (acc1: (Int, Int), acc2: (Int, Int)) => (acc1._1 + acc2._1, acc1._2 + acc2._2)
).map{ case (key, value) => (key, value._1 / value._2.toFloat)}
```

The result of running previous code in Spark Shell:

*Code Listing 54: Spark combineByKey Result in Spark Shell*

```
scala> pairAverageResult.take(5);

res6: Array[(Int, Float)] = Array((1,1.5), (2,3.5), (3,0.0))
```

We are just trying stuff in Spark Shell; the simplest method to partially debug RDDs is to take the first  $n$  elements out of them. We know that we only have three keys in collection, so taking five out will definitely print them all out. The purpose of these examples is not so much on performance and production level optimization, but more that you get comfortable with using Spark API until the end of the book. We covered a pretty complex example in Scala. Let's have a look at how this would look in Python:

*Code Listing 55: Spark combineByKey Python code*

```
pair_average_result = pairs_rdd.combineByKey((lambda v: (v, 1)),
  (lambda a1, v: (a1[0] + v, a1[1] + 1)),
  (lambda a2, a1: (a2[0] + a1[0], a2[1] + a1[1]))
).map(lambda acc: (acc[0], (acc[1][0] / float(acc[1][1]))))
```

Now that we have the code, let's have a look at the result:

*Code Listing 56: Spark combineByKey Result in PySpark*

```
>>> pair_average_result.take(5)

[(1, 1.5), (2, 3.5), (3, 0.0)]
```

The Scala and Python examples are relatively similar. Now we are going to show the Java 7 version. You will notice there is much more code when compared to Scala and Python. For completeness sake, let's have a look at complete code:

*Code Listing 57: Spark combineByKey Java7 Code*

```
package com.syncfusion.succinctly.spark.examples;
```

```

import java.io.PrintWriter;
import java.io.Serializable;
import java.util.Arrays;
import java.util.List;
import java.util.Map;

import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.api.java.function.Function;
import org.apache.spark.api.java.function.Function2;

import scala.Tuple2;

public class CombineByKeyJava7 {

    public static class AverageCount implements Serializable {
        public int _sum;
        public int _count;

        public AverageCount(int sum, int count) {
            this._sum = sum;
            this._count = count;
        }

        public float average() {
            return _sum / (float) _count;
        }
    }

    public static void main(String[] args) throws Exception {
        if (args.length < 1) {
            System.err.println("Please provide a full path to the output file");
            System.exit(0);
        }

        SparkConf conf = new
SparkConf().setAppName("CombineByKeyJava7").setMaster("local");
        JavaSparkContext context = new JavaSparkContext(conf);

        Function<Integer, AverageCount> createCombiner =
            new Function<Integer, AverageCount>() {
                public AverageCount call(Integer x) {
                    return new AverageCount(x, 1);
                }
            };

```



```

        Function2<AverageCount, Integer, AverageCount> mergeValues =
            new Function2<AverageCount, Integer, AverageCount>() {
                public AverageCount call(AverageCount a, Integer x) {
                    a._sum += x;
                    a._count += 1;
                    return a;
                }
            };

        Function2<AverageCount, AverageCount, AverageCount> mergeTwoCombiners =
            new Function2<AverageCount, AverageCount, AverageCount>() {
                public AverageCount call(AverageCount a, AverageCount b) {
                    a._sum += b._sum;
                    a._count += b._count;
                    return a;
                }
            };

        List<Tuple2<Integer, Integer>> pairs = Arrays.asList(new Tuple2(1, 1), new
        Tuple2<Integer, Integer>(1, 2),
                    new Tuple2<Integer, Integer>(2, 3), new Tuple2<Integer,
        Integer>(2, 4),
                    new Tuple2<Integer, Integer>(3, 0));

        JavaPairRDD<Integer, Integer> pairsRDD = context.parallelizePairs(pairs);

        JavaPairRDD<Integer, AverageCount> pairAverageResult =
        pairsRDD.combineByKey(createCombiner, mergeValues,
                    mergeTwoCombiners);

        Map<Integer, AverageCount> resultMap = pairAverageResult.collectAsMap();

        PrintWriter writer = new PrintWriter(args[0]);

        for (Integer key : resultMap.keySet()) {
            writer.println(key + ", " + resultMap.get(key).average());
        }

        writer.close();
    }
}

```

Code should produce an output file according to parameters you give it when running. In my case, it was written to the combineByKey.txt file and it looked like:

*Code Listing 58: Spark combineByKey Java 7 Version in combineByKey.txt*

```
2, 3.5
1, 1.5
3, 0.0
```

Again, this seems like a lot of code, but I guess Java veterans are used to it—perhaps not liking it, but that’s the way it is with pre-Java 8 versions. Just as a comparison, we’ll have a look at the algorithmic part with Java 8:

*Code Listing 59: Spark combineByKey Java 8 Algorithmic Part*

```
JavaPairRDD<Integer, AverageCount> pairAverageResult = pairsRDD.combineByKey(
    x -> new AverageCount(x, 1),
    (a, x) -> { a._sum += x; a._count += 1; return a; },
    (a, b) -> { a._sum += b._sum; a._count += b._count; return a; });
```

Note that imports, parameter processing, and class definition remain as in the previous example. Java examples have more code because at the moment Java doesn’t support an interactive shell. The submit task will also be pretty similar to the Java 7 example. You will probably just give your class a different name. All in all, there is a lot less code in this example as you can see. Since there is less code, it’s probably even easier to maintain it.

In this section, we demonstrated the basics of data aggregation. Again, there is more to it than shown here, but the goal of this book is to get you going. In the next section we are going to deal with data grouping.

## Data Grouping

Grouping is very similar to aggregation. The difference is that it only groups data; there is no computing of values per group as in aggregation. Grouping can be used as a basis for other operations, like join and intersect between two Pair RDDs, but doing that is usually a very bad idea because it is not very efficient. In the previous section we used simple integer key value pairs. Examples in this section wouldn’t make much sense if we did that, so we are going to define simple data sets from the three languages that we are using in this book. To go over examples for this section, we are going to define a simple example about used cars. The inside records will be a year, a name, and a color. We will also create a Pair RDD. Let’s start with the Scala version and then do the same for Python and Java:

*Code Listing 60: Scala Used Cars Initial Data Definition*

```
case class UsedCar(year: Int, name: String, color: String)

val cars = List(
    UsedCar(2012, "Honda Accord LX", "gray"),
    UsedCar(2008, "Audi Q7 4.2 Premium", "white"),
    UsedCar(2006, "Mercedes-Benz CLS-Class CLS500", "black"),
```

```

UsedCar(2012, "Mitsubishi Lancer", "black"),
UsedCar(2010, "BMW 3 Series 328i", "gray"),
UsedCar(2006, "Buick Lucerne CX", "gray"),
UsedCar(2006, "GMC Envoy", "red")
)

```

*Code Listing 61: Python Used Cars Initial Data Definition*

```

cars = [
    (2012, 'Honda Accord LX', 'gray'),
    (2008, 'Audi Q7 4.2 Premium', 'white'),
    (2006, 'Mercedes-Benz CLS-Class CLS500', 'black'),
    (2012, 'Mitsubishi Lancer', 'black'),
    (2010, 'BMW 3 Series 328i', 'gray'),
    (2006, 'Buick Lucerne CX', 'gray'),
    (2006, 'GMC Envoy', 'red')
]

```

*Code Listing 62: Java Used Cars Initial Data Definition*

```

public static class UsedCar implements Serializable {
    public int _year;
    public String _name;
    public String _color;

    public UsedCar(int year, String name, String color) {
        this._year = year;
        this._name = name;
        this._color = color;
    }
}

// rest of the code coming soon

List<UsedCar> cars = Arrays.asList(
    new UsedCar(2012, "Honda Accord LX", "gray"),
    new UsedCar(2008, "Audi Q7 4.2 Premium", "white"),
    new UsedCar(2006, "Mercedes-Benz CLS-Class CLS500", "black"),
    new UsedCar(2012, "Mitsubishi Lancer", "black"),
    new UsedCar(2010, "BMW 3 Series 328i", "gray"),
    new UsedCar(2006, "Buick Lucerne CX", "gray"),
    new UsedCar(2006, "GMC Envoy", "red")
);

```

We'll create Pair RDDs with year being a key and the entry itself as a value:

*Code Listing 63: Scala Pair RDD with Year as a Key and Entry as a Value*

```
val carsByYear = sc.parallelize(cars).map(x => (x.year, x))
```

*Code Listing 64: Python Pair RDD with Year as a Key and Entry as a Value*

```
cars_by_year = sc.parallelize(cars).map(lambda x : (x[0], x))
```

With Java we would have to repeat a lot of code here, so we'll just skip it for now and provide the final example at the end. Our goal now is to fetch all used car offerings for a year. We'll use year 2006 in our examples. To do this, we are going to use **groupBy** method. Most of the time, **reduceByKey** and **combineByKey** have better performance, but **groupBy** comes in very handy when we need to group the data together. Note that grouping transformations may result in big pairs. Remember that any given Pair RDD element must fit into memory. RDD can split to disk, but a single Pair RDD instance must fit into memory. Let's group the dataset by year and then we'll look up all the values for 2006 in Scala:

*Code Listing 65: Scala Used Cars from 2006*

```
scala> carsByYear.groupByKey().lookup(2006)

res6: Seq[Iterable[UsedCar]] = ArrayBuffer(CompactBuffer(UsedCar(2006,Mercedes-Benz CLS-Class CLS500,black), UsedCar(2006,Buick Lucerne CX,gray), UsedCar(2006,GMC Envoy,red)))
```

With Python we will show how the whole data structure looks, just so that you get the overall impression:

*Code Listing 66: Python Showing All Available Used Cars Grouped by Years*

```
>>> cars_by_year.groupByKey().map(lambda x : (x[0], list(x[1]))).collect()

[(2008, [(2008, 'Audi Q7 4.2 Premium', 'white')]), (2012, [(2012, 'Honda Accord LX', 'gray'), (2012, 'Mitsubishi Lancer', 'black')]), (2010, [(2010, 'BMW 3 Series 328i', 'gray')]), (2006, [(2006, 'Mercedes-Benz CLS-Class CLS500', 'black'), (2006, 'Buick Lucerne CX', 'gray'), (2006, 'GMC Envoy', 'red')])]
```

*Code Listing 67: Java Example for groupByKey*

```
package com.syncfusion.succinctly.spark.examples;

import java.io.PrintWriter;
import java.io.Serializable;
import java.util.Arrays;
import java.util.List;

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

```

import org.apache.spark.api.java.function.PairFunction;

import scala.Tuple2;

public class GroupByKeyJava7 {

    public static class UsedCar implements Serializable {
        public int _year;
        public String _name;
        public String _color;

        public UsedCar(int year, String name, String color) {
            this._year = year;
            this._name = name;
            this._color = color;
        }
    }

    public static void main(String[] args) throws Exception {
        if (args.length < 1) {
            System.err.println("Please provide a full path to the output file");
            System.exit(0);
        }

        SparkConf conf = new SparkConf()
            .setAppName("GroupByKeyJava7").setMaster("local");
        JavaSparkContext context = new JavaSparkContext(conf);

        List<UsedCar> cars = Arrays.asList(
            new UsedCar(2012, "Honda Accord LX", "gray"),
            new UsedCar(2008, "Audi Q7 4.2 Premium", "white"),
            new UsedCar(2006, "Mercedes-Benz CLS-Class CLS500", "black"),
            new UsedCar(2012, "Mitsubishi Lancer", "black"),
            new UsedCar(2010, "BMW 3 Series 328i", "gray"),
            new UsedCar(2006, "Buick Lucerne CX", "gray"),
            new UsedCar(2006, "GMC Envoy", "red")
        );

        JavaPairRDD<Integer, UsedCar> carsByYear = context.parallelize(cars)
            .mapToPair(new PairFunction<UsedCar, Integer, UsedCar>() {
                public Tuple2<Integer, UsedCar> call(UsedCar c) {
                    return new Tuple2<Integer, UsedCar>(c._year, c);
                }
            });

        List<Iterable<UsedCar>> usedCars2006 = carsByYear.groupByKey().lookup(2006);
    }
}

```

```

        PrintWriter writer = new PrintWriter(args[0]);

        for (Iterable<UsedCar> usedCars : usedCars2006) {
            for (UsedCar usedCar : usedCars) {
                writer.println(
                    usedCar._year
                    + ", " + usedCar._name
                    + ", " + usedCar._color);
            }
        }

        writer.close();
    }
}

```

The result of running the Java example is a file containing offerings for cars from 2006:

*Code Listing 68: Result of Java Example Processing*

```

$ cat /syncfusion/spark/out/groupByKey.txt

2006, Mercedes-Benz CLS-Class CLS500, black
2006, Buick Lucerne CX, gray
2006, GMC Envoy, red

```

With this example, we went over how to group data together in Spark. In the next section we are going to go over sorting pair RDDs.

## Sorting Pair RDDs

Sorting is a very important feature in everyday computer applications. People perceive data and notice important differences and trends much better if data is sorted. In many situations, the sorting happens by time. Other requirements, like sorting alphabetically, are totally valid. There is one important fact that you have to remember:



**Note:** *Sorting is an expensive operation. Try to sort smaller sets.*

When we sort, all of the machines in the cluster have to contact each other and exchange multiple data sets over and over again throughout the time it takes to sort the data. So, when you need sorting, try to invoke it on smaller data sets. We will use data defined in the previous section. In this section, we will simply sort it by year:

*Code Listing 69: Scala Sorting Car Offerings by Year Descending*

```
scala> carsByYear.sortByKey(false).collect().foreach(println)

(2012,UsedCar(2012,Honda Accord LX,gray))
(2012,UsedCar(2012,Mitsubishi Lancer,black))
(2010,UsedCar(2010,BMW 3 Series 328i,gray))
(2008,UsedCar(2008,Audi Q7 4.2 Premium,white))
(2006,UsedCar(2006,Mercedes-Benz CLS-Class CLS500,black))
(2006,UsedCar(2006,Buick Lucerne CX,gray))
(2006,UsedCar(2006,GMC Envoy,red))
```

*Code Listing 70: Python Sorting Car Offerings by Year*

```
cars_sorted_by_years = cars_by_year.sortByKey().map(lambda x : (x[0], list(x[1]))).collect()

for x in cars_sorted_by_years:
    print str(x[0]) + ', ' + x[1][1] + ', ' + x[1][2]
```

*Code Listing 71: Python Results of Running in PySpark*

```
>>> for x in cars_sorted_by_years:
...     print str(x[0]) + ', ' + x[1][1] + ', ' + x[1][2]
...
2006, Mercedes-Benz CLS-Class CLS500, black
2006, Buick Lucerne CX, gray
2006, GMC Envoy, red
2008, Audi Q7 4.2 Premium, white
2010, BMW 3 Series 328i, gray
2012, Honda Accord LX, gray
2012, Mitsubishi Lancer, black
>>>
```

The Java example will again contain a bit more code due to reasons we've already mentioned:

*Code Listing 72: Code for Sorting by Keys in Java*

```
package com.syncfusion.succinctly.spark.examples;

import java.io.PrintWriter;
import java.io.Serializable;
import java.util.Arrays;
import java.util.List;

import org.apache.spark.SparkConf;
```

```

import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.api.java.function.PairFunction;

import scala.Tuple2;

public class SortByKeyJava7 {

    public static class UsedCar implements Serializable {
        public int _year;
        public String _name;
        public String _color;

        public UsedCar(int year, String name, String color) {
            this._year = year;
            this._name = name;
            this._color = color;
        }
    }

    public static void main(String[] args) throws Exception {
        if (args.length < 1) {
            System.err.println("Please provide a full path to the output file");
            System.exit(0);
        }

        SparkConf conf = new
SparkConf().setAppName("SortByKeyJava7").setMaster("local");
        JavaSparkContext context = new JavaSparkContext(conf);

        List<UsedCar> cars = Arrays.asList(
            new UsedCar(2012, "Honda Accord LX", "gray"),
            new UsedCar(2008, "Audi Q7 4.2 Premium", "white"),
            new UsedCar(2006, "Mercedes-Benz CLS-Class CLS500", "black"),
            new UsedCar(2012, "Mitsubishi Lancer", "black"),
            new UsedCar(2010, "BMW 3 Series 328i", "gray"),
            new UsedCar(2006, "Buick Lucerne CX", "gray"),
            new UsedCar(2006, "GMC Envoy", "red")
        );

        JavaPairRDD<Integer, UsedCar> carsByYear = context.parallelize(cars)
            .mapToPair(new PairFunction<UsedCar, Integer, UsedCar>() {
                public Tuple2<Integer, UsedCar> call(UsedCar c) {
                    return new Tuple2<Integer, UsedCar>(c._year, c);
                }
            });
    }
}

```



```

        List<Tuple2<Integer, UsedCar>> sortedCars = carsByYear.sortByKey().collect();

        PrintWriter writer = new PrintWriter(args[0]);

        for (Tuple2<Integer, UsedCar> usedCar : sortedCars) {
            writer.println(
                usedCar._2()._year
                + ", " + usedCar._2()._name
                + ", " + usedCar._2()._color);
        }

        writer.close();
    }
}

```

The results of Java processing are:

*Code Listing 73: Results of Java Code Sorting*

```

$ cat /syncfusion/spark/out/sortByKey.txt

2006, Mercedes-Benz CLS-Class CLS500, black
2006, Buick Lucerne CX, gray
2006, GMC Envoy, red
2008, Audi Q7 4.2 Premium, white
2010, BMW 3 Series 328i, gray
2012, Honda Accord LX, gray
2012, Mitsubishi Lancer, black

```

There is one more important group of operations that we can do with Pair RDDs. It's joining, and we'll discuss it in the next section

## Join, Intersect, Union, and Difference Operations on Pair RDDs

One of the most important Pair RDD operations is joining. It's especially useful when working with structured data, which we'll discuss later. There are three types of join operations available: inner, left, and right. Note that joins are relatively expensive operations in Spark because they tend to move data around the cluster. This doesn't mean that you shouldn't use them at all, just be careful and expect longer processing times if you are joining really big data sets together. Here is the overview of join operations:

Table 6: Join operations

Transformation	Description
<code>join(otherDataset)</code>	When called on datasets (K, V) and (K, W), it returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key.
<code>leftOuterJoin(otherDataset)</code>	When called on datasets (K, V) and (K, W), it returns a dataset with all pairs (K, (V, Some(W))). W is of type Optional, and we'll show in our example how to work with Optionals.
<code>rightOuterJoin(otherDataset)</code>	When called on datasets (K, V) and (K, W), it returns a dataset with all pairs (K, Some(V), W). V is of type Optional.

We'll continue with the used car market example when working with Pair RDDs. We'll add information about car dealers to our previous dataset. Here is the dataset that we're going to use for Scala and Python. We won't go through every step for Java. For Java we'll provide source code after Scala and Python examples. The data definition is as following:

Code Listing 74: Scala Used Cars and Dealers Example

```
case class UsedCar(dealer: String, year: Int, name: String, color: String)

case class Dealer(dealer: String, name: String)

val cars = List(
  UsedCar("a", 2012, "Honda Accord LX", "gray"),
  UsedCar("b", 2008, "Audi Q7 4.2 Premium", "white"),
  UsedCar("c", 2006, "Mercedes-Benz CLS-Class CLS500", "black"),
  UsedCar("a", 2012, "Mitsubishi Lancer", "black"),
  UsedCar("c", 2010, "BMW 3 Series 328i", "gray"),
  UsedCar("c", 2006, "Buick Lucerne CX", "gray"),
  UsedCar("c", 2006, "GMC Envoy", "red"),
  UsedCar("unknown", 2006, "GMC Envoy", "red")
)

val carsByDealer = sc.parallelize(cars).map(x => (x.dealer, x))

val dealers = List(
  Dealer("a", "Top cars"),
  Dealer("b", "Local dealer"),
  Dealer("d", "Best cars")
)

var dealersById = sc.parallelize(dealers).map(x => (x.dealer, x))
```

*Code Listing 75: Python Used Cars and Dealers Example*

```
cars = [
    ("a", 2012, "Honda Accord LX", "gray"),
    ("b", 2008, "Audi Q7 4.2 Premium", "white"),
    ("c", 2006, "Mercedes-Benz CLS-Class CLS500", "black"),
    ("a", 2012, "Mitsubishi Lancer", "black"),
    ("c", 2010, "BMW 3 Series 328i", "gray"),
    ("c", 2006, "Buick Lucerne CX", "gray"),
    ("c", 2006, "GMC Envoy", "red"),
    ("unknown", 2006, "GMC Envoy", "red")
]

cars_by_dealer = sc.parallelize(cars).map(lambda x: (x[0], x))

dealers = [
    ("a", "Top cars"),
    ("b", "Local dealer"),
    ("d", "Best cars")
]

dealers_by_id = sc.parallelize(dealers).map(lambda x: (x[0], x))
```

Now that we have defined our data, we can go over join operations in Scala and Python. Let's start with regular inner join operation:

*Code Listing 76: Scala Dealers and Cars Join Operation in Spark Shell*

```
scala> carsByDealer.join(dealersById).foreach(println)

(a,(UsedCar(a,2012,Honda Accord LX,gray),Dealer(a,Top cars)))
(a,(UsedCar(a,2012,Mitsubishi Lancer,black),Dealer(a,Top cars)))
(b,(UsedCar(b,2008,Audi Q7 4.2 Premium,white),Dealer(b,Local dealer)))
```

*Code Listing 77: Python Dealers and Cars Join Operation in PySpark*

```
>>> cars_and_dealers_joined = cars_by_dealer.join(dealers_by_id)
>>>
>>> for x in cars_and_dealers_joined.collect():
...     print x[0] + ', ' + str(x[1][0][1]) + ', ' + x[1][0][2] + ', ' + x[1][1][1]
...

a, 2012, Honda Accord LX, Top cars
a, 2012, Mitsubishi Lancer, Top cars
b, 2008, Audi Q7 4.2 Premium, Local dealer
```

Join works by creating elements only if they are present on both sides. Sometimes we need to display information regardless of their presence on both sides. In that case we use left and right join. Here is an example of using **leftOuterJoin**:

*Code Listing 78: Scala Dealers and Cars Left Outer Join Operation in Spark Shell*

```
scala> carsByDealer.leftOuterJoin(dealersById).foreach(println)

(b,(UsedCar(b,2008,Audi Q7 4.2 Premium,white),Some(Dealer(b,Local dealer))))
(unknown,(UsedCar(unknown,2006,GMC Envoy,red),None))
(c,(UsedCar(c,2006,Mercedes-Benz CLS-Class CLS500,black),None))
(c,(UsedCar(c,2010,BMW 3 Series 328i,gray),None))
(c,(UsedCar(c,2006,Buick Lucerne CX,gray),None))
(c,(UsedCar(c,2006,GMC Envoy,red),None))
(a,(UsedCar(a,2012,Honda Accord LX,gray),Some(Dealer(a,Top cars))))
(a,(UsedCar(a,2012,Mitsubishi Lancer,black),Some(Dealer(a,Top cars))))
```

*Code Listing 79: Python Dealers and Cars Left Outer Join Operation in PySpark*

```
cars_and_dealers_joined_left = cars_by_dealer.leftOuterJoin(dealers_by_id)

for x in cars_and_dealers_joined_left.collect():
    out = x[0] + ', ' + str(x[1][0][1]) + ', ' + x[1][0][2] + ', '
    if x[1][1] is None:
        out += 'None'
    else:
        out += x[1][1][1]
    print out
...
a, 2012, Honda Accord LX, Top cars
a, 2012, Mitsubishi Lancer, Top cars
unknown, 2006, GMC Envoy, None
c, 2006, Mercedes-Benz CLS-Class CLS500, None
c, 2010, BMW 3 Series 328i, None
c, 2006, Buick Lucerne CX, None
c, 2006, GMC Envoy, None
b, 2008, Audi Q7 4.2 Premium, Local dealer
```

Right joining would then look something like:

*Code Listing 80: Scala Dealers and Cars Right Outer Join Operation in Spark Shell*

```
scala> carsByDealer.rightOuterJoin(dealersById).foreach(println)

(d,(None,Dealer(d,Best cars)))
(b,(Some(UsedCar(b,2008,Audi Q7 4.2 Premium,white)),Dealer(b,Local dealer)))
(a,(Some(UsedCar(a,2012,Honda Accord LX,gray)),Dealer(a,Top cars)))
```

```
(a,(Some(UsedCar(a,2012,Mitsubishi Lancer,black)),Dealer(a,Top cars)))
```

*Code Listing 81: Python Dealers and Cars Right Outer Join Operation in Spark Shell*

```
cars_and_dealers_joined_right = cars_by_dealer.rightOuterJoin(dealers_by_id)

for x in cars_and_dealers_joined_right.collect():
    out = x[0] + ', '
    if x[1][0] is None:
        out += 'None'
    else:
        out += str(x[1][0][1]) + ', ' + x[1][0][2] + ', '
    out += x[1][1][1]
    print out
...

a, 2012, Honda Accord LX, Top cars
a, 2012, Mitsubishi Lancer, Top cars
b, 2008, Audi Q7 4.2 Premium, Local dealer
d, None, Best cars
```

With Java, we are just going to have one big example with all the operations at the end. We'll use code comments to mark every important block there. There are other important Pair RDD operations besides joins. One of them is intersection. Intersection is not defined as a separate operation on RDD, but it can be expressed by using other operations. Here is what it would look like in Scala:

*Code Listing 82: Scala Dealers and Cars Intersection Operation in Spark Shell*

```
scala> carsByDealer.groupByKey().join(dealersById.groupByKey())
.flatMapValues(t => t._1++t._2).foreach(println)

(a,UsedCar(a,2012,Honda Accord LX,gray))
(b,UsedCar(b,2008,Audi Q7 4.2 Premium,white))
(b,Dealer(b,Local dealer))
(a,UsedCar(a,2012,Mitsubishi Lancer,black))
(a,Dealer(a,Top cars))
```

*Code Listing 83: Python Dealers and Cars Intersection Operation in PySpark*

```
cars_and_dealers_intersection =
cars_by_dealer.groupByKey().join(dealers_by_id.groupByKey()).flatMapValues(lambda x: [x[0],
x[1]])

for x in cars_and_dealers_intersection.collect():
```

```

for xi in x[1]:
    print x[0] + ', ' + str(xi[1]) + (' ' + xi[2] if len(xi) > 2 else '')
...
a, 2012, Honda Accord LX
a, 2012, Mitsubishi Lancer
unknown, 2006, GMC Envoy
c, 2006, Mercedes-Benz CLS-Class CLS500
c, 2010, BMW 3 Series 328i
c, 2006, Buick Lucerne CX
c, 2006, GMC Envoy
b, 2008, Audi Q7 4.2 Premium

```

Note that usually intersection, union, and difference are combined with the same type of data. The intent for previous examples was to demonstrate that everything remains flexible in Spark. That and combining real-life entities is better than joining some generic data like numbers. But if you try to call an actual union operator on two collections, you would get an error saying that the types are not compatible. Here is an example of the actual Spark Shell call:

*Code Listing 84: Fail of Scala Dealers and Cars Union Operation in Spark Shell*

```

scala> carsByDealer.union(dealersById).foreach(println)

<console>:34: error: type mismatch;
found   : org.apache.spark.rdd.RDD[(String, Dealer)]
required: org.apache.spark.rdd.RDD[(String, UsedCar)]
carsByDealer.union(dealersById).foreach(println)

```

To continue with the examples, we are going to define another group of cars. We will simply give suffix 2 to this group:

*Code Listing 85: Scala Second Group of Cars*

```

val cars2 = List(
  UsedCar("a", 2008, "Toyota Auris", "black"),
  UsedCar("c", 2006, "GMC Envoy", "red")
)

val carsByDealer2 = sc.parallelize(cars2).map(x => (x.dealer, x))

```

*Code Listing 86: Python Second Group of Cars*

```

cars2 = [
    ("a", 2008, "Toyota Auris", "black"),
    ("c", 2006, "GMC Envoy", "red")
]

cars_by_dealer2 = sc.parallelize(cars2).map(lambda x: (x[0], x))

```

Now that we have the second group defined, we can have a look at what the union operation is doing:

*Code Listing 87: Scala Union Operation Example*

```
scala> carsByDealer.union(carsByDealer2).foreach(println)

(c,UsedCar(c,2006,Mercedes-Benz CLS-Class CLS500,black))
(a,UsedCar(a,2012,Mitsubishi Lancer,black))
(a,UsedCar(a,2012,Honda Accord LX,gray))
(b,UsedCar(b,2008,Audi Q7 4.2 Premium,white))
(c,UsedCar(c,2010,BMW 3 Series 328i,gray))
(c,UsedCar(c,2006,Buick Lucerne CX,gray))
(c,UsedCar(c,2006,GMC Envoy,red))
(unknown,UsedCar(unknown,2006,GMC Envoy,red))
(a,UsedCar(a,2008,Toyota Auris,black))
(c,UsedCar(c,2006,GMC Envoy,red))
```

*Code Listing 88: Python Union Operation Example*

```
cars_union = cars_by_dealer.union(cars_by_dealer2)

for x in cars_union.collect():
    print str(x[0]) + ', ' + str(x[1][1]) + ', ' + x[1][2]
...
a, 2012, Honda Accord LX
b, 2008, Audi Q7 4.2 Premium
c, 2006, Mercedes-Benz CLS-Class CLS500
a, 2012, Mitsubishi Lancer
c, 2010, BMW 3 Series 328i
c, 2006, Buick Lucerne CX
c, 2006, GMC Envoy
unknown, 2006, GMC Envoy
a, 2008, Toyota Auris
c, 2006, GMC Envoy
```

The last operation that we are going to discuss is difference. Difference is computed by calling the **subtractByKey** operation:

*Code Listing 89: Scala Difference Operation*

```
scala> carsByDealer.subtractByKey(carsByDealer2).foreach(println)

(b,UsedCar(b,2008,Audi Q7 4.2 Premium,white))
(unknown,UsedCar(unknown,2006,GMC Envoy,red))
```

*Code Listing 90: Python Difference Operation*

```
cars_difference = cars_by_dealer.subtractByKey(cars_by_dealer2)

for x in cars_difference.collect():
    print str(x[0]) + ', ' + str(x[1][1]) + ', ' + x[1][2]
...
unknown, 2006, GMC Envoy
b, 2008, Audi Q7 4.2 Premium
```

This completes the overview of operations between two Pair RDDs. The Java examples are much more verbose, so I decided to group them into one example to save code on initialization. Here is the Java version of code for the entire section.

*Code Listing 91: Operations between Two Pair RDDs in Java*

```
package com.syncfusion.succinctly.spark.examples;

import java.io.Serializable;
import java.util.Arrays;
import java.util.List;

import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.api.java.function.PairFunction;

import com.google.common.base.Optional;
import scala.Tuple2;

public class OperationsBetweenToPairRddsJava7 {

    public static class UsedCar implements Serializable {
        public String _dealer;
        public int _year;
        public String _name;
        public String _color;

        public UsedCar(String dealer, int year, String name, String color) {
            this._dealer = dealer;
            this._year = year;
            this._name = name;
            this._color = color;
        }

        @Override
        public String toString() {
```



```

        return _dealer + ", " + _year + ", " + _name + ", " + _color;
    }
}

public static class Dealer implements Serializable {
    public String _dealer;
    public String _name;

    public Dealer(String dealer, String name) {
        this._dealer = dealer;
        this._name = name;
    }

    @Override
    public String toString() {
        return _dealer + ", " + _name;
    }
}

public static void main(String[] args) throws Exception {
    if (args.length < 1) {
        System.err.println("Please provide a full path to the output file");
        System.exit(0);
    }

    SparkConf conf = new
SparkConf().setAppName("OperationsBetweenToPairRddsJava7").setMaster("local");
    JavaSparkContext context = new JavaSparkContext(conf);

    List<UsedCar> cars = Arrays.asList(
        new UsedCar("a", 2012, "Honda Accord LX", "gray"),
        new UsedCar("b", 2008, "Audi Q7 4.2 Premium", "white"),
        new UsedCar("c", 2006, "Mercedes-Benz CLS-Class CLS500",
"black"),

        new UsedCar("a", 2012, "Mitsubishi Lancer", "black"),
        new UsedCar("c", 2010, "BMW 3 Series 328i", "gray"),
        new UsedCar("c", 2006, "Buick Lucerne CX", "gray"),
        new UsedCar("c", 2006, "GMC Envoy", "red"),
        new UsedCar("unknown", 2006, "GMC Envoy", "red"));

    PairFunction<UsedCar, String, UsedCar> mapCars = new PairFunction<UsedCar,
String, UsedCar>() {
        public Tuple2<String, UsedCar> call(UsedCar c) {
            return new Tuple2<String, UsedCar>(c._dealer, c);
        }
    };
};

```

```

        JavaPairRDD<String, UsedCar> carsByDealer =
context.parallelize(cars).mapToPair(mapCars);

        List<Dealer> dealers = Arrays.asList(
            new Dealer("a", "Top cars"),
            new Dealer("b", "Local dealer"),
            new Dealer("d", "Best cars")

        );

        JavaPairRDD<String, Dealer> dealersById = context.parallelize(dealers)
            .mapToPair(new PairFunction<Dealer, String, Dealer>() {
                public Tuple2<String, Dealer> call(Dealer c) {
                    return new Tuple2<String, Dealer>(c._dealer, c);
                }
            });

        // join operation
        JavaPairRDD<String, Tuple2<UsedCar, Dealer>> carsAndDealersJoined =
carsByDealer.join(dealersById);
        carsAndDealersJoined.saveAsTextFile(args[0] + "carsAndDealersJoined");

        // left outer join operation
        JavaPairRDD<String, Tuple2<UsedCar, Optional<Dealer>>>
carsAndDealersJoinedLeft = carsByDealer
            .leftOuterJoin(dealersById);
        carsAndDealersJoinedLeft.saveAsTextFile(args[0] + "carsAndDealersJoinedLeft");

        // right outer join operation
        JavaPairRDD<String, Tuple2<Optional<UsedCar>, Dealer>>
carsAndDealersJoinedRight = carsByDealer
            .rightOuterJoin(dealersById);
        carsAndDealersJoinedLeft.saveAsTextFile(args[0] + "carsAndDealersJoinedRight");

        // intersection
        JavaPairRDD<String, Tuple2<Iterable<UsedCar>, Iterable<Dealer>>>
carsAndDealersIntersection = carsByDealer
            .groupByKey().join(dealersById.groupByKey());
        carsAndDealersJoinedLeft.saveAsTextFile(args[0] + "carsAndDealersIntersection");

        List<UsedCar> cars2 = Arrays.asList(new UsedCar("a", 2008, "Toyota Auris",
"black"),
            new UsedCar("c", 2006, "GMC Envoy", "red")

        );

```

```

        JavaPairRDD<String, UsedCar> carsByDealer2 =
context.parallelize(cars).mapToPair(mapCars);

        // union
        JavaPairRDD<String, UsedCar> carsUnion = carsByDealer.union(carsByDealer2);
        carsUnion.saveAsTextFile(args[0] + "carsUnion");

        // difference
        JavaPairRDD<String, UsedCar> carsDifference =
carsByDealer.subtractByKey(carsByDealer2);
        carsUnion.saveAsTextFile(args[0] + "carsDifference");
    }
}

```

As practice for this section, you can go to the output directory that you specified when running the submit task for this example and have a look at the output files. See if you can notice anything interesting.

In this section we went over the operations between two Pair RDDs in Spark. It's a very important subject when doing distributed computing and processing data with Spark. This concludes the chapter about Spark's internal mechanism. In the next chapter we are going to discuss how Spark acquires the data, and we'll focus on how to deliver results data to channels other than text files and interactive shells.

# Chapter 4 Data Input and Output with Spark

By now, we have gone over a fair share of concepts and recipes on the subject of data processing in Spark. Up until now we mainly output data into text files on disk or simply used interactive shells for Scala and Python. The interactive shell and regular text files are perfectly fine when it comes to learning, and there is nothing wrong with this approach. But sooner or later you will come across a situation where you will have to read data from some standard input source or store it in a way that other processes in the environment will be able to use it. Spark supports a lot of input and output capabilities. We won't go over every single one of them but will mention the most important ones. In this chapter we are also going to cover how Spark can store data into some of the popular storage technologies like Cassandra. Let's start with simple text files.

## Working with Text Files

We have worked with some examples where we used text files as input and output for our examples. Just as a small refresher, let's have a look at how to make RDDs representing every line in a text file:

*Code Listing 92: Text File Loaded as RDD in Scala*

```
scala> val input = sc.textFile("README.md")

input: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[31] at textFile at <console>:21
```

*Code Listing 93: Text file Loaded as RDD in Python*

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

*Code Listing 94: Text File Loaded as RDD in Java*

```
JavaRDD<String> inputFile = context.textFile("README.md");
```

We went over how we could use the created RDD in word count examples earlier in the book, so we don't need to go into details right now. However, there is one very useful feature. Namely, sometimes the data won't fit into a single file. For instance, if we have really large logs where a new log file is created every now and then, we could use wildcards like "\*" to read them all into RDD at once. All at once doesn't literally mean "go over the whole file at once and load it into memory;" remember, RDDs are lazily loaded, so until a call to action there is no reading of data. Just to make sure that we are on a same page, we are going to analyze a simple file that we create. This will be the number of sold used cars per day. We won't complicate anything; we'll just create two files. Place them on your file system to a folder of your choosing:

*Code Listing 95: Example File sold-units-2015-08.txt*

```
1000
2000
1000
5000
4000
10000
11000
300
5500
```

*Code Listing 96: Example File sold-units-2015-09.txt*

```
4000
5000
6000
6000
3000
4000
500
700
8800
```

Now we are going to make example programs that analyze the data and compute average sold cars in two months. Let's start with Scala:

*Code Listing 97: Scala Example for Reading Sold Units*

```
val salesRaw = sc.wholeTextFiles("sold-units-2015*")

val sales = salesRaw.map(x => x._2).
  flatMap(x => x.split("\n")).
  map(x => x.toDouble)
```

Note that elements of RDD store file description and file content. We are interested only in the content part, so we map by the second part of a tuple. Let's say that we are interested in an average. Up until now, we haven't mentioned that spark supports arithmetic functions on RDDs that contain only doubles in them. Calculating averages is actually really easy in Spark:

*Code Listing 98: Scala Showing Average Price of Sold Unit*

```
scala> sales.mean()

res30: Double = 4322.222222222223
```

Let's do the same with Python:

*Code Listing 99: Python Example for Reading Sold Units*

```
sales_raw = sc.wholeTextFiles("sold-units-2015*")

sales = sales_raw.map(lambda x: x[1]
).flatMap(lambda x: x.split("\n")
).map(lambda x: float(x))
```

*Code Listing 100: Python Showing Average Price of Sold Unit*

```
>>> sales.mean()

4322.222222222223
```

Let's have a look at the Java version:

*Code Listing 101: Java Example for Average Price of Sold Unit*

```
package com.syncfusion.succinctly.spark.examples;

import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.util.Arrays;

import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaDoubleRDD;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaSparkContext;
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 scala.Tuple2;

public class WholeTextFileMeanUnitPriceJava7 {

    public static void main(String[] args) throws FileNotFoundException {
        if (args.length < 1) {
            System.err.println("Please provide a full path to the input files");
            System.exit(0);
        }

        if (args.length < 2) {
            System.err.println("Please provide a full path to the output file");
            System.exit(0);
        }
    }
}
```

```

        SparkConf conf = new SparkConf()
            .setAppName("WholeTextFileJava7")
            .setMaster("local");
        JavaSparkContext context = new JavaSparkContext(conf);

        JavaPairRDD<String, String> salesRaw = context.wholeTextFiles(args[0]);

        JavaDoubleRDD sales = salesRaw.map(new Function<Tuple2<String, String>,
String>() {
            public String call(Tuple2<String, String> v1) throws Exception {
                return v1._2();
            }
        }).flatMap(new FlatMapFunction<String, String>() {
            public Iterable<String> call(String t) throws Exception {
                return Arrays.asList(t.split("\n"));
            }
        }).mapToDouble(new DoubleFunction<String>() {
            @Override
            public double call(String t) throws Exception {
                return Double.parseDouble(t);
            }
        });

        PrintWriter writer = new PrintWriter(args[1]);
        writer.println(sales.mean());
        writer.close();
    }
}

```

Be careful when submitting tasks to Spark. We want all files having a pattern to be resolved by Spark and not by the shell. Here is how I submitted java task to spark:

*Code Listing 102 – Task Submit Operation on Linux*

```

$ ./bin/spark-submit \
--class com.synCFusion.succinctly.spark.examples. WholeTextFileMeanUnitPriceJava7 \
--master local \
/root/spark/SparkExamples-0.0.1-SNAPSHOT.jar \
"/root/spark-1.4.1-bin-hadoop2.6/sold-units*" /root/spark/result.txt

$ cat /root/spark/result.txt

4322.222222222223

```

You might run into trouble if you submit the task without quotations because output will go to one of the input files and you will get no feedback on running. You will also overwrite the input file.

## Submitting Scala and Python Tasks to Spark

Until now we haven't needed to submit Python and Scala tasks to Spark directly. We simply used the interactive shells and were fine. We only showed how to submit Java applications to Spark because Java, at the moment, does not have an interactive shell. In my opinion, it's best to acquire information when you need it because otherwise you would probably just skim over the section in earlier parts of the book and completely forget about it. Now you will need it. In this chapter we are going to go over JSON serialization in the context of Spark, so we need to include some libraries in our project. We will make a simple word count application again. First, you have to create a file system directory and file structure like this one:

*Code Listing 103: Basic Scala Project Structure*

```
$ tree
.
├── simple.sbt
└── src
    ├── main
    │   └── scala
    │       └── ExampleApp.scala
```

*Code Listing 104: File Content of simple.sbt File*

```
name := "Example application"

version := "1.0"

scalaVersion := "2.10.5"

libraryDependencies += "org.apache.spark" %% "spark-core" % "1.4.1"
```

*Code Listing 105: File Content of ExampleApp.scala File*

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

object ExampleApp {
  def main(args: Array[String]) {
    val conf = new SparkConf().setAppName("ExampleApp")
    val sc = new SparkContext(conf)

    val readmeFile = sc.textFile(args(0))
```



```

val sparkMentions = readmeFile.filter(line => line.contains("Spark"))

scala.tools.nsc.io.File(args(1)).writeAll(sparkMentions.count().toString)
}
}

```

When you are done creating directory structure and editing files, run the package command. This will create a **.jar** file. The process is very similar to building a Java application:

*Code Listing 106: Running Package Command to Build .jar File*

```
$ sbt package
```

This will result in creating the **./target/scala-2.10/example-application\_2.10-1.0.jar** file. Names can vary if you change versions. Submitting a task to Spark is then pretty similar to submitting a Java task:

*Code Listing 107: Submitting Scala Example to Spark*

```

$ ./bin/spark-submit \
--class ExampleApp \
--master local \
/root/spark/scala/ExampleApp/target/scala-2.10/example-application_2.10-1.0.jar \
/root/spark-1.4.1-bin-hadoop2.6/README.md /root/spark/scala_example.txt

```

Submitting a task to Spark is relatively simple with Python. The first part is creating a Python script somewhere on your file system. Let's call it something simple like **simple.py**:

*Code Listing 108: Python Script simple.py*

```

import sys

from PySpark import SparkContext

sc = SparkContext(appName="ExampleApp")

readme_file = sc.textFile(sys.argv[1])
spark_mentions = readme_file.filter(lambda line: "Spark" in line)

out_file = open(sys.argv[2], 'w+')
out_file.write(str(spark_mentions.count()))

```

Running the example against Spark is then relatively easy:

*Code Listing 109: Python Script simple.py*

```
$ ./bin/spark-submit \  
--master local \  
/root/spark/python/simple.py \  
/root/spark-1.4.1-bin-hadoop2.6/README.md /root/spark/python_example.txt
```

In the next section, we are going to go over techniques for working with JSON.

## Working with JSON files

JSON is one of the most popular data formats of today. It's used everywhere, from configuration files to web page communication. That's one of the main reasons why we are going to cover how to use it with Spark. One of the simplest methods of JSON loading is by treating external data as text and then mapping it by using a library. The same goes for writing it out. We'll create a simple JSON file to use in our example:

*Code Listing 110: Simple File with Multiple JSON Entries test.json*

```
{"year": 2012, "name": "Honda Accord LX", "color": "gray"}  
{"year": 2008, "name": "Audi Q7 4.2 Premium", "color": "white"}  
{"year": 2006, "name": "Mercedes-Benz CLS-Class CLS500", "color": "black"}  
{"year": 2012, "name": "Mitsubishi Lancer", "color": "black"}  
{"year": 2010, "name": "BMW 3 Series 328i", "color": "gray"}  
{"year": 2006, "name": "Buick Lucerne CX", "color": "gray"}  
{"year": 2006, "name": "GMC Envoy", "color": "red"}
```

Let's load this with Scala and then sort the data by years:

*Code Listing 111: Scala Code to Load and Sort test.json*

```
import org.apache.spark.SparkContext  
import org.apache.spark.SparkContext._  
import org.apache.spark.SparkConf  
  
import com.fasterxml.jackson.module.scala._  
import com.fasterxml.jackson.databind.ObjectMapper  
import com.fasterxml.jackson.databind.DeserializationFeature  
  
import java.io.StringWriter;  
  
object JsonLoading {  
  case class UsedCar(year: Int, name: String, color: String) extends Serializable  
  
  def main(args: Array[String]) {  
    val conf = new SparkConf().setAppName("JsonLoading")
```

```

val sc = new SparkContext(conf)

// mapper can't get serialized by spark at the moment
// using workaround
object Holder extends Serializable {
  @transient lazy val mapper = new ObjectMapper()
  mapper.registerModule(DefaultScalaModule)
}

val inputFile = sc.textFile(args(0))
val usedCars = inputFile.map(x => Holder.mapper.readValue(x, classOf[UsedCar]))

val carsByYear = usedCars.map(x => (x.year, x))

val carsSortedByYear = carsByYear.sortByKey(false).collect()

val out = new StringWriter()
Holder.mapper.writeValue(out, carsSortedByYear)
val json = out.toString()

scala.tools.nsc.io.File(args(1)).writeAll(json)
}
}

```

The resulting file looks like this:

*Code Listing 112: Resulting JSON File*

```

[
  [
    2012,
    {
      "year": 2012,
      "name": "Honda Accord LX",
      "color": "gray"
    }
  ],
  [
    2012,
    {
      "year": 2012,
      "name": "Mitsubishi Lancer",
      "color": "black"
    }
  ],
  [

```



*Code Listing 113: Python Script to Sort the Entries in JSON*

```
import sys
import json

from PySpark import SparkContext

sc = SparkContext(appName="JsonLoading")

input_file = sc.textFile(sys.argv[1])
used_cars = input_file.map(lambda x: json.loads(x))

cars_by_year = used_cars.map(lambda x: (x["year"], x))

cars_sorted_by_year = cars_by_year.sortByKey(False)

mapped = cars_sorted_by_year.map(lambda x: json.dumps(x))

mapped.saveAsTextFile(sys.argv[2])
```

After submitting the Python script you should get the following output:

*Code Listing 114: Result of Submitting Python Script*

```
[2012, {"color": "gray", "name": "Honda Accord LX", "year": 2012}]
[2012, {"color": "black", "name": "Mitsubishi Lancer", "year": 2012}]
[2010, {"color": "gray", "name": "BMW 3 Series 328i", "year": 2010}]
[2008, {"color": "white", "name": "Audi Q7 4.2 Premium", "year": 2008}]
[2006, {"color": "black", "name": "Mercedes-Benz CLS-Class CLS500", "year": 2006}]
[2006, {"color": "gray", "name": "Buick Lucerne CX", "year": 2006}]
[2006, {"color": "red", "name": "GMC Envoy", "year": 2006}]
```

As a final example in this section let's have a look at how we could do JSON manipulations with Java:

*Code Listing 115: Java Manipulating JSON*

```
package com.syncfusion.succinctly.spark.examples;

import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.io.Serializable;
import java.util.List;

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

```

import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.api.java.function.Function;
import org.apache.spark.api.java.function.PairFunction;

import com.fasterxml.jackson.core.JsonProcessingException;
import com.fasterxml.jackson.databind.ObjectMapper;

import scala.Tuple2;

public class JsonManipulationJava7 {

    public static class UsedCar implements Serializable {
        public int year;
        public String name;
        public String color;

        // for object mapping
        public UsedCar() {

        }

        public UsedCar(int year, String name, String color) {
            this.year = year;
            this.name = name;
            this.color = color;
        }
    }

    public static void main(String[] args) throws FileNotFoundException,
    JsonProcessingException {
        if (args.length < 1) {
            System.err.println("Please provide a full path to the input files");
            System.exit(0);
        }

        if (args.length < 2) {
            System.err.println("Please provide a full path to the output file");
            System.exit(0);
        }

        SparkConf conf = new SparkConf()
            .setAppName("JsonManipulationJava7")
            .setMaster("local");
        JavaSparkContext sc = new JavaSparkContext(conf);

        JavaRDD<String> inputFile = sc.textFile(args[0]);
    }
}

```

```

        ObjectMapper mapper = new ObjectMapper();

        JavaRDD<UsedCar> usedCars = inputFile.map(new Function<String, UsedCar>() {
            public UsedCar call(String v1) throws Exception {
                return mapper.readValue(v1, UsedCar.class);
            }
        });

        JavaPairRDD<Integer, UsedCar> carsByYear = usedCars.mapToPair(new
        PairFunction<UsedCar, Integer, UsedCar>() {
            @Override
            public Tuple2<Integer, UsedCar> call(UsedCar t) throws Exception {
                return new Tuple2<Integer, UsedCar>(t.year, t);
            }
        });

        List<Tuple2<Integer, UsedCar>> carsSortedByYear =
        carsByYear.sortByKey(false).collect();

        PrintWriter writer = new PrintWriter(args[1]);
        writer.println(mapper.writeValueAsString(carsSortedByYear));
        writer.close();
    }
}

```

Java also demonstrates exporting a tuple to JSON format. Just so that you can see what it looks like:

*Code Listing 116: Result of Java JSON Manipulation*

```

[
  {
    "_1": 2012,
    "_2": {
      "year": 2012,
      "name": "Honda Accord LX",
      "color": "gray"
    }
  },
  {
    "_1": 2012,
    "_2": {
      "year": 2012,
      "name": "Mitsubishi Lancer",
      "color": "black"
    }
  }
]

```

```

    }
  },
  {
    "_1": 2010,
    "_2": {
      "year": 2010,
      "name": "BMW 3 Series 328i",
      "color": "gray"
    }
  },
  {
    "_1": 2008,
    "_2": {
      "year": 2008,
      "name": "Audi Q7 4.2 Premium",
      "color": "white"
    }
  },
  {
    "_1": 2006,
    "_2": {
      "year": 2006,
      "name": "Mercedes-Benz CLS-Class CLS500",
      "color": "black"
    }
  },
  {
    "_1": 2006,
    "_2": {
      "year": 2006,
      "name": "Buick Lucerne CX",
      "color": "gray"
    }
  },
  {
    "_1": 2006,
    "_2": {
      "year": 2006,
      "name": "GMC Envoy",
      "color": "red"
    }
  }
]

```



There are a lot of other supported formats, including CSV, Sequence Files, and more. We won't go into them since we just went over the most basic ones, but there is one more very important thing that we haven't discussed. It's how Spark interacts with databases. Spark has very good support for interacting with relational databases. Spark is also often described as a big-data processing framework. So perhaps it would be best to concentrate on one of the most popular big-data storage technologies of today. It's Apache Cassandra, and it works very well with Spark.

## Spark and Cassandra

Spark and Cassandra are a natural fit: big-data processing framework on one side and big-data storage technology on the other. Now, to be honest, I might be a little biased because I work with Cassandra a lot, and my personal opinion is that it's better to talk about the things you know. I would also like to make a complete, end-to-end example using Spark and Cassandra.

It's best that we start by installing Cassandra. In previous chapters we described how to install Java on your computer. That's more than enough to start making things with Cassandra. The next step for you is to download Cassandra itself. There is a commercial version available from a company called DataStax, but to get you started the community version is just fine. Head over to <http://www.planetcassandra.org/cassandra/> and download the appropriate version for your system. I won't go into much detail as it's really out of scope for this book, but you can find everything you need under <http://www.planetcassandra.org/try-cassandra/>. There are lots of tutorials available on how to set up Cassandra, but as I mentioned earlier the most important prerequisite is an installed version of Java. If you like using Syncfusion's resources, there are a lot of free books available at <http://www.syncfusion.com/resources/techportal/ebooks>, and *Cassandra Succinctly* is one of them. So without further ado, from this point on I'm assuming you have a running Cassandra instance on your local machine. This section might also be a bit more challenging than the previous ones, but I'll try to provide as much detail as possible.

One of Cassandra's strong points is that it can store a lot of data in a relatively small amount of time. This is all thanks to Cassandra's internals, at the core of which lies the concept of wide row. We won't go much into it from a theoretical point of view, but I think you will be able to follow along through examples. Cassandra currently has no support for doing aggregate operations like sum, average, and the like. The only available operation is count. Aggregate operations are huge performance killers, even in the relational world. Sooner or later there is one query that simply takes too long, and then administrators or engineers go into it, and what they most often do is create some sort of table or view that has the data prepared right away so that there is no need for joins. That's actually how you model data with Cassandra, but to be honest, it's a topic of its own. We will talk a little bit about internals. Unlike traditional databases, Cassandra can have up to two billion columns per single row. That's where the name wide row comes from. Cassandra keeps all those columns sorted all the time. You can access the values by sorted columns, but you can't calculate averages and similar things as with relational databases, so you have to use other technologies, such as Spark in our case.

Navigate to the folder where you installed Cassandra and run a tool called **cqlsh**. It stands for Cassandra Query Language Shell. We'll set up the data by using this shell, and then we'll process the data by using Spark and store the results of our analysis back to Spark. This is mostly introductory material, so I'll try to keep it simple. I won't always use the best production level practices because my goal is to get you started. If somebody with experience in the field is reading this, please don't hold it against me. The started **cqlsh** should look something like:

*Code Listing 117: Running CQL Shell*

```
$ ./cqlsh
Connected to Test Cluster at 127.0.0.1:9042.
[cqlsh 5.0.1 | Cassandra 2.1.8 | CQL spec 3.2.0 | Native protocol v3]
Use HELP for help.
cqlsh>
```

If you are running your cluster for the very first time, you don't have defined keyspaces and tables to store data. Don't think too much about keyspace; for now, you can think of it as a container for tables. Let's define one. In fact, let's define two of them right away. Spark will read data from one container and write them to the other one. There are usually multiple keyspaces in production. Note, however, that this is not a production level example:

*Code Listing 118: Definition of Keyspaces for Cassandra Weather Station Example*

```
CREATE KEYSPACE weather_in
WITH REPLICATION = {
  'class' : 'SimpleStrategy',
  'replication_factor' : 1
};

CREATE KEYSPACE weather_out
WITH REPLICATION = {
  'class' : 'SimpleStrategy',
  'replication_factor' : 1
};
```

Let's create a table for input data:

*Code Listing 119: Definition of Table for Cassandra Weather Station Example*

```
use weather_in;

CREATE TABLE measurements (
  station text,
  time timestamp,
  temperature decimal,
  PRIMARY KEY (station, time)
);
```

Primary key syntax works a bit different than in relational databases. Here, the first part of the primary key specifies what data will have a long row. In our case, every weather station will have one large row. The second part of primary key syntax specifies how data will be sorted in this long row. In our case, it will be sorted by time. Essentially, every time a reading comes in it goes to the end of a long row.

Let's define some data on our own. We don't care if that data is in Celsius or Fahrenheit, so we'll keep the values around zero so that it's possible on both scales. The difference will be cold and colder depending on which unit you prefer:

*Code Listing 120: Some Data (Imagine It Came from a Weather Station)*

```
INSERT INTO measurements (station, time, temperature) VALUES ('A', '2015-12-01 10:00:00', 1);
INSERT INTO measurements (station, time, temperature) VALUES ('A', '2015-12-01 12:00:00', 2);
INSERT INTO measurements (station, time, temperature) VALUES ('A', '2015-12-02 10:00:00', 3);
INSERT INTO measurements (station, time, temperature) VALUES ('A', '2015-12-02 12:00:00', 5);
INSERT INTO measurements (station, time, temperature) VALUES ('A', '2015-12-02 14:00:00', 5);

INSERT INTO measurements (station, time, temperature) VALUES ('B', '2015-12-01 10:00:00', 3);
INSERT INTO measurements (station, time, temperature) VALUES ('B', '2015-12-01 12:00:00', 4);
INSERT INTO measurements (station, time, temperature) VALUES ('B', '2015-12-02 10:00:00', 0);
INSERT INTO measurements (station, time, temperature) VALUES ('B', '2015-12-02 12:00:00', 2);
INSERT INTO measurements (station, time, temperature) VALUES ('B', '2015-12-02 14:00:00', 1);
```

You can use a star operator and select all of the data out. But it's not the best practice in production; Cassandra even has a default limit of 10,000 records so that you don't break anything in production. A correct way to access the data is by providing a station where the measurement took place. For simplicity, we only have stations A and B. If we wanted to know what the temperature was around a certain time on both station A and B, we would have no problem obtaining the data since measurements are sorted by time. But if we tried to select all of the measurements where temperature is above two degrees following, this would happen:

*Code Listing 121: Trying to Select All Temperatures Above Certain Value*

```
cqlsh:weather_in> SELECT * from measurements where station = 'A' AND temperature > 2;

InvalidRequest: code=2200 [Invalid query] message="No secondary indexes on the restricted
columns support the provided operators: "
```

And that's why we need Spark. Let's say we are interested in the all-time average at a weather station. We'll keep it simple so we just need to store a weather station, average and the exact time when average was calculated. We'll let Spark populate the output table. To connect to Cassandra, we are going to use Spark Cassandra Connector, available under <https://github.com/datastax/spark-cassandra-connector>. In this section we are only going to go over Scala code. You can look up the tutorial on Spark Cassandra Connector and do the example provided here in the language of your preference. The concepts that we cover here are enough to get you started.

We went over how to make stand-alone Spark applications a couple of chapters ago. To connect to Cassandra, we need to add new dependencies. The problem is that the dependencies also have to include some other dependencies. And if we wanted to submit a task to Spark, Spark would not be able to locate referenced .jar files. We'll have to adapt the project structure a little bit to enable building a so-called fat jar.

If we want to connect to Cassandra, we need to add a new dependency. Also, when we build this fat jar, it shouldn't include files already included in Spark. To achieve this goal we are going to use a very popular tool available at <https://github.com/sbt/sbt-assembly/>. We will have to modify the structure of the Spark example a bit. We have to create a new file **build.sbt** in the example's root directory. This file should have the following content:

*Code Listing 122: Create New build.sbt File in Scala Example Directory*

```
lazy val commonSettings = Seq(
  version := "0.1-SNAPSHOT",
  organization := "syncfusion",
  scalaVersion := "2.10.5"
)

lazy val app = (project in file("app")).
  settings(commonSettings: _*).
  settings(
  )
```

Update contents of **simple.sbt** file:

*Code Listing 123: Create New simple.sbt File in Scala Example Directory*

```
name := "example-application"

version := "1.0"

scalaVersion := "2.10.5"

libraryDependencies += "org.apache.spark" %% "spark-core" % "1.4.1" % "provided"
libraryDependencies += "com.datastax.spark" %% "spark-cassandra-connector" % "1.4.0-M3"
libraryDependencies += "com.datastax.spark" %% "spark-cassandra-connector-java" % "1.4.0-M3"
```

Create new Scala file:

*Code Listing 124: Source Code File CassandraProcessing.scala*

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

import com.datastax.spark.connector._
```

```

import com.datastax.driver.core._

import java.util.Date

object CassandraProcessing {
  case class Average(station: String, time: Date, average: BigDecimal)

  def main(args: Array[String]) {
    val conf = new SparkConf()
      .setAppName("CassandraProcessing")
      .set("spark.cassandra.connection.host", "127.0.0.1")

    val sc = new SparkContext(conf)

    val measurements = sc.cassandraTable("weather_in", "measurements")

    val measurementPairs = measurements.map(row => (row.getString("station"),
      row.getDecimal("temperature")))

    val averages = measurementPairs.mapValues(x => (x, 1)).reduceByKey((x, y) => (x._1 + y._1, x._2
      + y._2))

    var averagesForCassandra = averages.map(x => new Average(x._1, new Date(), x._2._1 / x._2._2))

    averagesForCassandra.saveToCassandra("weather_out", "all_time_average")
  }
}

```

Create the **assembly.sbt** file in the project folder with the following content:

*Code Listing 125: File project/assembly.sbt*

```
addSbtPlugin("com.eed3si9n" % "sbt-assembly" % "0.13.0")
```

Now create a fat jar by running the assembly command:

*Code Listing 126: Running Assembly Command*

```
$ sbt assembly
```

Watch the output of the assembly script and remember the location of the generated jar. Output will be different than in previous examples. Here is how my submit task looks:

*Code Listing 127: Submit Task*

```
$ ./bin/spark-submit \
```

```
--class CassandraProcessing \  
--master local \  
/root/spark/scala/ExampleApp/target/scala-2.10/example-application-assembly-1.0.jar
```

If everything went fine, you should see the results of processing in Cassandra.

*Code Listing 128: Results of Processing in Cassandra*

```
cqlsh> use weather_out;  
cqlsh> select * from all_time_average;
```

station	time	average
B	2015-08-17 22:12:25+0200	2
A	2015-08-17 22:12:25+0200	3.2

(2 rows)

In previous code listings, the table contained the results of the all-time average calculation and a timestamp when this average was computed. This example contains all the basic techniques that you will use if you get the chance to combine two technologies in practice. The provided example is filled with patterns that can be applied to all sorts of tasks.

Note that although Spark and Cassandra are relatively easy technologies to learn, when it comes to production they have a lot of best practices that you have to be aware of. In a sense, we have just scratched the surface here. But even the longest journey starts with the first step, and I hope that this section will bring you to a totally new path in your life.

With this section we covered a very important milestone on your path to Spark mastery. Be aware that Spark has a very vibrant community, and that you can develop yourself further by looking up online materials and visiting Spark conferences and meetups. I hope you enjoyed this e-book and the provided examples. Let's proceed to the conclusion.

# Conclusion

Spark is, at the moment, a star of the big data processing world. One could say it's getting more and more popular by the day. It's the most active open source big-data processing framework. A lot of this popularity is coming from a revolutionary idea at the heart of Spark, the idea that nodes in clusters, not just the disk, can share memory to process tasks.

One of Spark's selling points is that it can process data up to one hundred times faster than other open source big data technologies available today. This is truly incredible and important progress. Through human history, every now and then a technology comes by that changes the world. Some of them turn it upside down; some of them give a small contribution that proves very important in the long run. To be honest I don't know how to classify Spark, but it definitely changed a lot in the field of big data processing. Although Spark has been around for almost six years now, it has become incredibly popular by setting a record in the 2014 Daytona Gray Sort Challenge and beating Hadoop's previous record by running on one tenth of instances. Until that challenge, I had only heard about Spark every now and then and didn't know much about it. But when I heard about the record I started to look into it. That, and most of the cool people that I follow on Twitter started talking about it and combining it with Cassandra (and those two technologies go very well together). My feeling is that at the moment there are not that many Spark developers out there, so this book is more about an introduction to Spark along with a few more advanced topics. I felt I could contribute the most if I oriented myself towards Spark newcomers.

Recently I watched a talk by Brian Clapper on Spark. One of his statements was that the battle of NoSQL storage technologies is pretty much over, and the market is becoming pretty defined in that segment. In essence, past years yielded a solution for big systems to store all the data coming in. What I found important in his talk was he mentioned that processing wars are starting. After years of Hadoop dominating the world of data processing, new and exciting times are coming again. I hope this book provided you with solid foundations and opened the door to this exciting time.