

Jeffrey Aven

Sams **Teach Yourself**

# Apache Spark™

in **24**  
**Hours**

**SAMS**

FREE SAMPLE CHAPTER

SHARE WITH OTHERS



Jeffrey Aven

Sams **Teach Yourself**

# Apache Spark™

in **24**  
**Hours**

**SAMS**

800 East 96th Street, Indianapolis, Indiana, 46240 USA

## **Sams Teach Yourself Apache Spark™ in 24 Hours**

Copyright © 2017 by Pearson Education, Inc.

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

ISBN-13: 978-0-672-33851-9

ISBN-10: 0-672-33851-3

Library of Congress Control Number: 2016946659

Printed in the United States of America

First Printing: August 2016

### **Trademarks**

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Sams Publishing cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

### **Warning and Disclaimer**

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an “as is” basis. The author and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

### **Special Sales**

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at [corpsales@pearsoned.com](mailto:corpsales@pearsoned.com) or (800) 382-3419.

For government sales inquiries, please contact  
[governmentsales@pearsoned.com](mailto:governmentsales@pearsoned.com).

For questions about sales outside the U.S., please contact  
[intlcs@pearsoned.com](mailto:intlcs@pearsoned.com).

#### **Editor in Chief**

Greg Wiegand

#### **Acquisitions Editor**

Trina McDonald

#### **Development Editor**

Chris Zahn

#### **Technical Editor**

Cody Koeninger

#### **Managing Editor**

Sandra Schroeder

#### **Project Editor**

Lori Lyons

#### **Project Manager**

Ellora Sengupta

#### **Copy Editor**

Linda Morris

#### **Indexer**

Cheryl Lenser

#### **Proofreader**

Sudhakaran

#### **Editorial Assistant**

Olivia Basegio

#### **Cover Designer**

Chuti Prasertsith

#### **Compositor**

codeMantra

# Contents at a Glance

Preface .....	xii
About the Author .....	xv

## Part I: Getting Started with Apache Spark

<b>HOURL 1</b> Introducing Apache Spark .....	1
<b>2</b> Understanding Hadoop .....	11
<b>3</b> Installing Spark .....	27
<b>4</b> Understanding the Spark Application Architecture .....	45
<b>5</b> Deploying Spark in the Cloud .....	61

## Part II: Programming with Apache Spark

<b>HOURL 6</b> Learning the Basics of Spark Programming with RDDs .....	91
<b>7</b> Understanding MapReduce Concepts .....	115
<b>8</b> Getting Started with Scala .....	137
<b>9</b> Functional Programming with Python .....	165
<b>10</b> Working with the Spark API (Transformations and Actions) .....	197
<b>11</b> Using RDDs: Caching, Persistence, and Output .....	235
<b>12</b> Advanced Spark Programming .....	259

## Part III: Extensions to Spark

<b>HOURL 13</b> Using SQL with Spark .....	283
<b>14</b> Stream Processing with Spark .....	323
<b>15</b> Getting Started with Spark and R .....	343
<b>16</b> Machine Learning with Spark .....	363
<b>17</b> Introducing Sparkling Water (H2O and Spark) .....	381
<b>18</b> Graph Processing with Spark .....	399
<b>19</b> Using Spark with NoSQL Systems .....	417
<b>20</b> Using Spark with Messaging Systems .....	433

## **Part IV: Managing Spark**

<b>HOURL 21</b>	Administering Spark .....	453
<b>22</b>	Monitoring Spark .....	479
<b>23</b>	Extending and Securing Spark .....	501
<b>24</b>	Improving Spark Performance .....	519
	Index .....	543

# Table of Contents

Preface	xii
About the Author	xv

## Part I: Getting Started with Apache Spark

<b>HOOR 1: Introducing Apache Spark</b>	<b>1</b>
What Is Spark? .....	1
What Sort of Applications Use Spark? .....	3
Programming Interfaces to Spark .....	3
Ways to Use Spark .....	5
Summary .....	7
Q&A .....	8
Workshop .....	8
<b>HOOR 2: Understanding Hadoop</b>	<b>11</b>
Hadoop and a Brief History of Big Data .....	11
Hadoop Explained .....	12
Introducing HDFS .....	13
Introducing YARN .....	19
Anatomy of a Hadoop Cluster .....	22
How Spark Works with Hadoop .....	24
Summary .....	24
Q&A .....	25
Workshop .....	25
<b>HOOR 3: Installing Spark</b>	<b>27</b>
Spark Deployment Modes .....	27
Preparing to Install Spark .....	28
Installing Spark in Standalone Mode .....	29
Exploring the Spark Install .....	38

Deploying Spark on Hadoop .....	39
Summary .....	42
Q&A .....	43
Workshop .....	43
Exercises .....	44
<b>HOURL 4: Understanding the Spark Application Architecture</b> .....	<b>45</b>
Anatomy of a Spark Application .....	45
Spark Driver .....	46
Spark Executors and Workers .....	48
Spark Master and Cluster Manager .....	49
Spark Applications Running on YARN .....	51
Local Mode .....	56
Summary .....	58
Q&A .....	59
Workshop .....	59
<b>HOURL 5: Deploying Spark in the Cloud</b> .....	<b>61</b>
Amazon Web Services Primer .....	61
Spark on EC2 .....	64
Spark on EMR .....	73
Hosted Spark with Databricks .....	81
Summary .....	88
Q&A .....	89
Workshop .....	89
 <b>Part II: Programming with Apache Spark</b>	
<b>HOURL 6: Learning the Basics of Spark Programming with RDDs</b> .....	<b>91</b>
Introduction to RDDs .....	91
Loading Data into RDDs .....	93
Operations on RDDs .....	106
Types of RDDs .....	111
Summary .....	112
Q&A .....	113
Workshop .....	113

<b>HOOR 7: Understanding MapReduce Concepts</b>	<b>115</b>
MapReduce History and Background .....	115
Records and Key Value Pairs .....	117
MapReduce Explained .....	118
Word Count: The “Hello, World” of MapReduce .....	126
Summary .....	135
Q&A .....	135
Workshop .....	136
<b>HOOR 8: Getting Started with Scala</b>	<b>137</b>
Scala History and Background .....	137
Scala Basics .....	138
Object-Oriented Programming in Scala .....	153
Functional Programming in Scala .....	157
Spark Programming in Scala .....	160
Summary .....	163
Q&A .....	163
Workshop .....	163
<b>HOOR 9: Functional Programming with Python</b>	<b>165</b>
Python Overview .....	165
Data Structures and Serialization in Python .....	170
Python Functional Programming Basics .....	178
Interactive Programming Using IPython .....	183
Summary .....	193
Q&A .....	194
Workshop .....	194
<b>HOOR 10: Working with the Spark API (Transformations and Actions)</b>	<b>197</b>
RDDs and Data Sampling .....	197
Spark Transformations .....	199
Spark Actions .....	206
Key Value Pair Operations .....	211
Join Functions .....	219
Numerical RDD Operations .....	229



Summary .....	232
Q&A .....	232
Workshop .....	233
<b>HOURL 11: Using RDDs: Caching, Persistence, and Output</b>	<b>235</b>
RDD Storage Levels .....	235
Caching, Persistence, and Checkpointing .....	239
Saving RDD Output .....	247
Introduction to Alluxio (Tachyon) .....	254
Summary .....	257
Q&A .....	257
Workshop .....	258
<b>HOURL 12: Advanced Spark Programming</b>	<b>259</b>
Broadcast Variables .....	259
Accumulators .....	265
Partitioning and Repartitioning .....	270
Processing RDDs with External Programs .....	278
Summary .....	279
Q&A .....	280
Workshop .....	280
<b>Part III: Extensions to Spark</b>	
<b>HOURL 13: Using SQL with Spark</b>	<b>283</b>
Introduction to Spark SQL .....	283
Getting Started with Spark SQL DataFrames .....	294
Using Spark SQL DataFrames .....	305
Accessing Spark SQL .....	316
Summary .....	321
Q&A .....	321
Workshop .....	322
<b>HOURL 14: Stream Processing with Spark</b>	<b>323</b>
Introduction to Spark Streaming .....	323
Using DStreams .....	326

State Operations .....	335
Sliding Window Operations .....	337
Summary .....	339
Q&A .....	340
Workshop .....	340
<b>HOURL 15: Getting Started with Spark and R</b> .....	<b>343</b>
Introduction to R .....	343
Introducing SparkR .....	350
Using SparkR .....	355
Using SparkR with RStudio .....	358
Summary .....	360
Q&A .....	361
Workshop .....	361
<b>HOURL 16: Machine Learning with Spark</b> .....	<b>363</b>
Introduction to Machine Learning and MLlib .....	363
Classification Using Spark MLlib .....	367
Collaborative Filtering Using Spark MLlib .....	373
Clustering Using Spark MLlib .....	375
Summary .....	378
Q&A .....	378
Workshop .....	379
<b>HOURL 17: Introducing Sparkling Water (H2O and Spark)</b> .....	<b>381</b>
Introduction to H2O .....	381
Sparkling Water—H2O on Spark .....	387
Summary .....	396
Q&A .....	397
Workshop .....	397
<b>HOURL 18: Graph Processing with Spark</b> .....	<b>399</b>
Introduction to Graphs .....	399
Graph Processing in Spark .....	402
Introduction to GraphFrames .....	406

Summary .....	413
Q&A .....	414
Workshop.....	414
<b>HOOR 19: Using Spark with NoSQL Systems</b>	<b>417</b>
Introduction to NoSQL.....	417
Using Spark with HBase.....	419
Using Spark with Cassandra .....	425
Using Spark with DynamoDB and More.....	429
Summary .....	431
Q&A .....	431
Workshop.....	432
<b>HOOR 20: Using Spark with Messaging Systems</b>	<b>433</b>
Overview of Messaging Systems.....	433
Using Spark with Apache Kafka .....	435
Spark, MQTT, and the Internet of Things .....	443
Using Spark with Amazon Kinesis .....	446
Summary .....	450
Q&A .....	451
Workshop.....	451
<b>Part IV: Managing Spark</b>	
<b>HOOR 21: Administering Spark</b>	<b>453</b>
Spark Configuration.....	453
Administering Spark Standalone .....	461
Administering Spark on YARN .....	471
Summary .....	477
Q&A .....	477
Workshop.....	478
<b>HOOR 22: Monitoring Spark</b>	<b>479</b>
Exploring the Spark Application UI .....	479
Spark History Server .....	488
Spark Metrics .....	490

Logging in Spark .....	492
Summary .....	498
Q&A .....	499
Workshop .....	499
<b>HOURL 23: Extending and Securing Spark</b> .....	<b>501</b>
Isolating Spark .....	501
Securing Spark Communication .....	504
Securing Spark with Kerberos .....	512
Summary .....	516
Q&A .....	517
Workshop .....	517
<b>HOURL 24: Improving Spark Performance</b> .....	<b>519</b>
Benchmarking Spark .....	519
Application Development Best Practices .....	526
Optimizing Partitions .....	534
Diagnosing Application Performance Issues .....	536
Summary .....	540
Q&A .....	540
Workshop .....	541
<b>Index</b> .....	<b>543</b>

# Preface

This book assumes nothing, unlike many big data (Spark and Hadoop) books before it, which are often shrouded in complexity and assume years of prior experience. I don't assume that you are a seasoned software engineer with years of experience in Java, I don't assume that you are an experienced big data practitioner with extensive experience in Hadoop and other related open source software projects, and I don't assume that you are an experienced data scientist.

By the same token, you will not find this book patronizing or an insult to your intelligence either. The only prerequisite to this book is that you are “comfortable” with Python. Spark includes several application programming interfaces (APIs). The Python API was selected as the basis for this book as it is an intuitive, interpreted language that is widely known and easily learned by those who haven't used it.

This book could have easily been titled *Sams Teach Yourself Big Data Using Spark* because this is what I attempt to do, taking it from the beginning. I will introduce you to Hadoop, MapReduce, cloud computing, SQL, NoSQL, real-time stream processing, machine learning, and more, covering all topics in the context of how they pertain to Spark. I focus on core Spark concepts such as the Resilient Distributed Dataset (RDD), interacting with Spark using the shell, implementing common processing patterns, practical data engineering/analysis approaches using Spark, and much more.

I was first introduced to Spark in early 2013, which seems like a short time ago but is a lifetime ago in the context of the Hadoop ecosystem. Prior to this, I had been a Hadoop consultant and instructor for several years. Before writing this book, I had implemented and used Spark in several projects ranging in scale from small to medium business to enterprise implementations. Even having substantial exposure to Spark, researching and writing this book was a learning journey for myself, taking me further into areas of Spark that I had not yet appreciated. I would like to take you on this journey as well as you read this book.

Spark and Hadoop are subject areas I have dedicated myself to and that I am passionate about. The making of this book has been hard work but has truly been a labor of love. I hope this book launches your career as a big data practitioner and inspires you to do amazing things with Spark.

## Why Should I Learn Spark?

Spark is one of the most prominent big data processing platforms in use today and is one of the most popular big data open source projects ever. Spark has risen from its roots in academia to Silicon Valley start-ups to proliferation within traditional businesses such as banking, retail, and telecommunications. Whether you are a data analyst, data engineer, data scientist, or data steward, learning Spark will help you to advance your career or embark on a new career in the booming area of big data.

## How This Book Is Organized

This book starts by establishing some of the basic concepts behind Spark and Hadoop, which are covered in Part I, “Getting Started with Apache Spark.” I also cover deployment of Spark both locally and in the cloud in Part I.

Part II, “Programming with Apache Spark,” is focused on programming with Spark, which includes an introduction to functional programming with both Python and Scala as well as a detailed introduction to the Spark core API.

Part III, “Extensions to Spark,” covers extensions to Spark, which include Spark SQL, Spark Streaming, machine learning, and graph processing with Spark. Other areas such as NoSQL systems (such as Cassandra and HBase) and messaging systems (such as Kafka) are covered here as well.

I wrap things up in Part IV, “Managing Spark,” by discussing Spark management, administration, monitoring, and logging as well as securing Spark.

## Data Used in the Exercises

Data for the Try It Yourself exercises can be downloaded from the book’s Amazon Web Services (AWS) S3 bucket (if you are not familiar with AWS, don’t worry—I cover this topic in the book as well). When running the exercises, you can use the data directly from the S3 bucket or you can download the data locally first (examples of both methods are shown). If you choose to download the data first, you can do so from the book’s download page at <http://sty-spark.s3-website-us-east-1.amazonaws.com/>.

## Conventions Used in This Book

Each hour begins with “What You’ll Learn in This Hour,” which provides a list of bullet points highlighting the topics covered in that hour. Each hour concludes with a “Summary” page summarizing the main points covered in the hour as well as “Q&A” and “Quiz” sections to help you consolidate your learning from that hour.

Key topics being introduced for the first time are typically *italicized* by convention. Most hours also include programming examples in numbered code listings. Where functions, commands, classes, or objects are referred to in text, they appear in `monospace` type.

Other asides in this book include the following:

#### NOTE

---

Content not integral to the subject matter but worth noting or being aware of.

---

#### TIP

---

##### **TIP Subtitle**

A hint or tip relating to the current topic that could be useful.

---

#### CAUTION

---

##### **Caution Subtitle**

Something related to the current topic that could lead to issues if not addressed.

---

### ▼ TRY IT YOURSELF

#### **Exercise Title**

An exercise related to the current topic including a step-by-step guide and descriptions of expected outputs.

# About the Author

**Jeffrey Aven** is a big data consultant and instructor based in Melbourne, Australia. Jeff has an extensive background in data management and several years of experience consulting and teaching in the areas of Hadoop, HBase, Spark, and other big data ecosystem technologies. Jeff has won accolades as a big data instructor and is also an accomplished consultant who has been involved in several high-profile, enterprise-scale big data implementations across different industries in the region.



# Dedication

*This book is dedicated to my wife and three children. I have been burning the candle at both ends during the writing of this book and I appreciate your patience and understanding...*

# Acknowledgments

Special thanks to Cody Koeninger and Chris Zahn for their input and feedback as editors. Also thanks to Trina McDonald and all of the team at Pearson for keeping me in line during the writing of this book!

# We Want to Hear from You

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

We welcome your comments. You can email or write to let us know what you did or didn't like about this book—as well as what we can do to make our books better.

*Please note that we cannot help you with technical problems related to the topic of this book.*

When you write, please be sure to include this book's title and author as well as your name and email address. We will carefully review your comments and share them with the author and editors who worked on the book.

E-mail: [feedback@sampublishing.com](mailto:feedback@sampublishing.com)

Mail: Sams Publishing  
ATTN: Reader Feedback  
800 East 96th Street  
Indianapolis, IN 46240 USA

## Reader Services

Visit our website and register this book at [informit.com/register](http://informit.com/register) for convenient access to any updates, downloads, or errata that might be available for this book.

*This page intentionally left blank*

# HOUR 3

## Installing Spark

---

### What You'll Learn in This Hour:

- ▶ What the different Spark deployment modes are
- ▶ How to install Spark in Standalone mode
- ▶ How to install and use Spark on YARN

Now that you've gotten through the heavy stuff in the last two hours, you can dive headfirst into Spark and get your hands dirty, so to speak.

This hour covers the basics about how Spark is deployed and how to install Spark. I will also cover how to deploy Spark on Hadoop using the Hadoop scheduler, YARN, discussed in Hour 2.

By the end of this hour, you'll be up and running with an installation of Spark that you will use in subsequent hours.

## Spark Deployment Modes

There are three primary deployment modes for Spark:

- ▶ Spark Standalone
- ▶ Spark on YARN (Hadoop)
- ▶ Spark on Mesos

*Spark Standalone* refers to the built-in or “standalone” scheduler. The term can be confusing because you can have a single machine or a multinode fully distributed cluster both running in Spark Standalone mode. The term “standalone” simply means it does not need an external scheduler.

With Spark Standalone, you can get up and running quickly with few dependencies or environmental considerations. Spark Standalone includes everything you need to get started.

Spark on YARN and Spark on Mesos are deployment modes that use the resource schedulers YARN and Mesos respectively. In each case, you would need to establish a working YARN or Mesos cluster prior to installing and configuring Spark. In the case of Spark on YARN, this typically involves deploying Spark to an existing Hadoop cluster.

I will cover Spark Standalone and Spark on YARN installation examples in this hour because these are the most common deployment modes in use today.

## Preparing to Install Spark

Spark is a cross-platform application that can be deployed on

- ▶ Linux (all distributions)
- ▶ Windows
- ▶ Mac OS X

Although there are no specific hardware requirements, general Spark instance hardware recommendations are

- ▶ 8 GB or more memory
- ▶ Eight or more CPU cores
- ▶ 10 gigabit or greater network speed
- ▶ Four or more disks in *JBOD* configuration (JBOD stands for “Just a Bunch of Disks,” referring to independent hard disks not in a RAID—or Redundant Array of Independent Disks—configuration)

Spark is written in Scala with programming interfaces in Python (PySpark) and Scala. The following are software prerequisites for installing and running Spark:

- ▶ Java
- ▶ Python (if you intend to use PySpark)

If you wish to use Spark with R (as I will discuss in **Hour 15**, “**Getting Started with Spark and R**”), you will need to install R as well. Git, Maven, or SBT may be useful as well if you intend on building Spark from source or compiling Spark programs.

If you are deploying Spark on YARN or Mesos, of course, you need to have a functioning YARN or Mesos cluster before deploying and configuring Spark to work with these platforms.

I will cover installing Spark in Standalone mode on a single machine on each type of platform, including satisfying all of the dependencies and prerequisites.

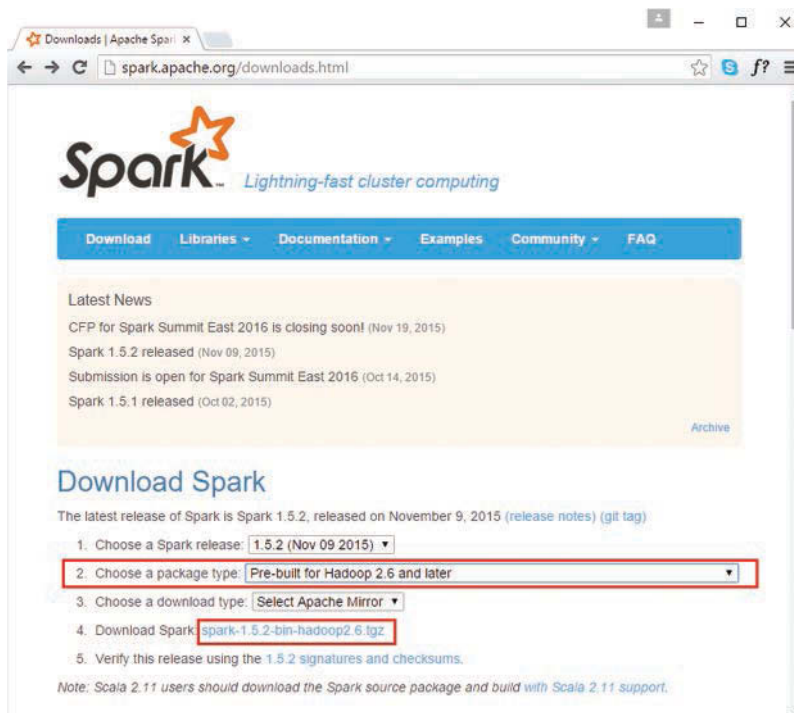
# Installing Spark in Standalone Mode

In this section I will cover deploying Spark in Standalone mode on a single machine using various platforms. Feel free to choose the platform that is most relevant to you to install Spark on.

## Getting Spark

In the installation steps for Linux and Mac OS X, I will use pre-built releases of Spark. You could also download the source code for Spark and build it yourself for your target platform using the build instructions provided on the official Spark website. I will use the latest Spark binary release in my examples. In either case, your first step, regardless of the intended installation platform, is to download either the release or source from: <http://spark.apache.org/downloads.html>

This page will allow you to download the latest release of Spark. In this example, the latest release is 1.5.2, your release will likely be greater than this (e.g. 1.6.x or 2.x.x).



**FIGURE 3.1**

The Apache Spark downloads page.

## NOTE

The Spark releases do not actually include Hadoop as the names may imply. They simply include libraries to integrate with the Hadoop clusters and distributions listed. Many of the Hadoop classes are required regardless of whether you are using Hadoop. I will use the `spark-1.5.2-bin-hadoop2.6.tgz` package for this installation.

## CAUTION

### Using the “Without Hadoop” Builds

You may be tempted to download the “without Hadoop” or `spark-x.x.x-bin-without-hadoop.tgz` options if you are installing in Standalone mode and not using Hadoop.

The nomenclature can be confusing, but this build is expecting many of the required classes that are implemented in Hadoop to be present on the system. Select this option only if you have Hadoop installed on the system already. Otherwise, as I have done in my case, use one of the `spark-x.x.x-bin-hadoopx.x` builds.

## ▼ TRY IT YOURSELF

### Install Spark on Red Hat/Centos

In this example, I’m installing Spark on a Red Hat Enterprise Linux 7.1 instance. However, the same installation steps would apply to Centos distributions as well.

1. As shown in Figure 3.1, download the `spark-1.5.2-bin-hadoop2.6.tgz` package from your local mirror into your home directory using `wget` or `curl`.
2. If Java 1.7 or higher is not installed, install the Java 1.7 runtime and development environments using the OpenJDK `yum` packages (alternatively, you could use the Oracle JDK instead):

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

3. Confirm Java was successfully installed:

```
$ java -version
java version "1.7.0_91"
OpenJDK Runtime Environment (rhel-2.6.2.3.el7-x86_64 u91-b00)
OpenJDK 64-Bit Server VM (build 24.91-b01, mixed mode)
```

4. Extract the Spark package and create `SPARK_HOME`:

```
tar -xzf spark-1.5.2-bin-hadoop2.6.tgz
sudo mv spark-1.5.2-bin-hadoop2.6 /opt/spark
export SPARK_HOME=/opt/spark
export PATH=$SPARK_HOME/bin:$PATH
```





## ▼ TRY IT YOURSELF

### Install Spark on Ubuntu/Debian Linux

In this example, I'm installing Spark on an Ubuntu 14.04 LTS Linux distribution.

As with the Red Hat example, Python 2.7 is already installed with the operating system, so we do not need to install Python.

1. As shown in Figure 3.1, download the `spark-1.5.2-bin-hadoop2.6.tgz` package from your local mirror into your home directory using `wget` or `curl`.
2. If Java 1.7 or higher is not installed, install the Java 1.7 runtime and development environments using Ubuntu's APT (Advanced Packaging Tool). Alternatively, you could use the Oracle JDK instead:

```
sudo apt-get update
sudo apt-get install openjdk-7-jre
sudo apt-get install openjdk-7-jdk
```

3. Confirm Java was successfully installed:

```
$ java -version
java version "1.7.0_91"
OpenJDK Runtime Environment (IcedTea 2.6.3) (7u91-2.6.3-0ubuntu0.14.04.1)
OpenJDK 64-Bit Server VM (build 24.91-b01, mixed mode)
```

4. Extract the Spark package and create `SPARK_HOME`:

```
tar -xzf spark-1.5.2-bin-hadoop2.6.tgz
sudo mv spark-1.5.2-bin-hadoop2.6 /opt/spark
export SPARK_HOME=/opt/spark
export PATH=$SPARK_HOME/bin:$PATH
```

The `SPARK_HOME` environment variable could also be set using the `.bashrc` file or similar user or system profile scripts. You will need to do this if you wish to persist the `SPARK_HOME` variable beyond the current session.

5. Open the PySpark shell by running the `pyspark` command from any directory. If Spark has been successfully installed, you should see the following output:

```
Welcome to
```

```

  ____
 /  __ \   _ __   _   _
 \  /  /  / __`/  / __`/
  \  /  /  / __`/  / __`/
   \  /  /  / __`/  / __`/
    \  /  /  / __`/  / __`/
     \  /  /  / __`/  / __`/
      \  /  /  / __`/  / __`/
       \  /  /  / __`/  / __`/
        \  /  /  / __`/  / __`/
         \  /  /  / __`/  / __`/
          \  /  /  / __`/  / __`/
           \  /  /  / __`/  / __`/
            \  /  /  / __`/  / __`/
             \  /  /  / __`/  / __`/
              \  /  /  / __`/  / __`/
               \  /  /  / __`/  / __`/
                \  /  /  / __`/  / __`/
                 \  /  /  / __`/  / __`/
                  \  /  /  / __`/  / __`/
                   \  /  /  / __`/  / __`/
                    \  /  /  / __`/  / __`/
                     \  /  /  / __`/  / __`/
                      \  /  /  / __`/  / __`/
                       \  /  /  / __`/  / __`/
                        \  /  /  / __`/  / __`/
                         \  /  /  / __`/  / __`/
                          \  /  /  / __`/  / __`/
                           \  /  /  / __`/  / __`/
                            \  /  /  / __`/  / __`/
                             \  /  /  / __`/  / __`/
                              \  /  /  / __`/  / __`/
                               \  /  /  / __`/  / __`/
                                \  /  /  / __`/  / __`/
                                 \  /  /  / __`/  / __`/
                                  \  /  /  / __`/  / __`/
                                   \  /  /  / __`/  / __`/
                                    \  /  /  / __`/  / __`/
                                     \  /  /  / __`/  / __`/
                                      \  /  /  / __`/  / __`/
                                       \  /  /  / __`/  / __`/
                                        \  /  /  / __`/  / __`/
                                         \  /  /  / __`/  / __`/
                                          \  /  /  / __`/  / __`/
                                           \  /  /  / __`/  / __`/
                                            \  /  /  / __`/  / __`/
                                             \  /  /  / __`/  / __`/
                                              \  /  /  / __`/  / __`/
                                               \  /  /  / __`/  / __`/
                                                \  /  /  / __`/  / __`/
                                                 \  /  /  / __`/  / __`/
                                                  \  /  /  / __`/  / __`/
                                                   \  /  /  / __`/  / __`/
                                                    \  /  /  / __`/  / __`/
                                                     \  /  /  / __`/  / __`/
                                                      \  /  /  / __`/  / __`/
                                                       \  /  /  / __`/  / __`/
                                                        \  /  /  / __`/  / __`/
                                                         \  /  /  / __`/  / __`/
                                                          \  /  /  / __`/  / __`/
                                                           \  /  /  / __`/  / __`/
                                                            \  /  /  / __`/  / __`/
                                                             \  /  /  / __`/  / __`/
                                                              \  /  /  / __`/  / __`/
                                                               \  /  /  / __`/  / __`/
                                                                \  /  /  / __`/  / __`/
                                                                 \  /  /  / __`/  / __`/
                                                                  \  /  /  / __`/  / __`/
                                                                   \  /  /  / __`/  / __`/
                                                                    \  /  /  / __`/  / __`/
                                                                     \  /  /  / __`/  / __`/
                                                                      \  /  /  / __`/  / __`/
                                                                       \  /  /  / __`/  / __`/
                                                                        \  /  /  / __`/  / __`/
                                                                         \  /  /  / __`/  / __`/
                                                                          \  /  /  / __`/  / __`/
                                                                           \  /  /  / __`/  / __`/
                                                                            \  /  /  / __`/  / __`/
                                                                             \  /  /  / __`/  / __`/
                                                                              \  /  /  / __`/  / __`/
                                                                               \  /  /  / __`/  / __`/
                                                                                \  /  /  / __`/  / __`/
                                                                                 \  /  /  / __`/  / __`/
                                                                                  \  /  /  / __`/  / __`/
                                                                                   \  /  /  / __`/  / __`/
                                                                                    \  /  /  / __`/  / __`/
                                                                                     \  /  /  / __`/  / __`/
                                                                                      \  /  /  / __`/  / __`/
                                                                                       \  /  /  / __`/  / __`/
                                                                                        \  /  /  / __`/  / __`/
                                                                                         \  /  /  / __`/  / __`/
                                                                                          \  /  /  / __`/  / __`/
                                                                                           \  /  /  / __`/  / __`/
                                                                                            \  /  /  / __`/  / __`/
                                                                                             \  /  /  / __`/  / __`/
                                                                                              \  /  /  / __`/  / __`/
                                                                                               \  /  /  / __`/  / __`/
                                                                                                \  /  /  / __`/  / __`/
                                                                                                 \  /  /  / __`/  / __`/
                                                                                                  \  /  /  / __`/  / __`/
                                                                                                   \  /  /  / __`/  / __`/
                                                                                                    \  /  /  / __`/  / __`/
                                                                                                     \  /  /  / __`/  / __`/
                                                                                                      \  /  /  / __`/  / __`/
                                                                                                       \  /  /  / __`/  / __`/
                                                                                                        \  /  /  / __`/  / __`/
                                                                                                         \  /  /  / __`/  / __`/
                                                                                                          \  /  /  / __`/  / __`/
                                                                                                           \  /  /  / __`/  / __`/
                                                                                                            \  /  /  / __`/  / __`/
                                                                                                             \  /  /  / __`/  / __`/
                                                                                                              \  /  /  / __`/  / __`/
                                                                                                               \  /  /  / __`/  / __`/
                                                                                                                \  /  /  / __`/  / __`/
                                                                                                                 \  /  /  / __`/  / __`/
                                                                                                                  \  /  /  / __`/  / __`/
                                                                                                                   \  /  /  / __`/  / __`/
                                                                                                                    \  /  /  / __`/  / __`/
                                                                                                                     \  /  /  / __`/  / __`/
                                                                                                                      \  /  /  / __`/  / __`/
                                                                                                                       \  /  /  / __`/  / __`/
                                                                                                                        \  /  /  / __`/  / __`/
                                                                                                                         \  /  /  / __`/  / __`/
                                                                                                                          \  /  /  / __`/  / __`/
                                                                                                                           \  /  /  / __`/  / __`/
                                                                                                                            \  /  /  / __`/  / __`/
                                                                                                                             \  /  /  / __`/  / __`/
                                                                                      version 1.5.2

```

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





4. You should see a similar result by running the `spark-shell` command in the terminal from any directory.

5. Run the included Pi Estimator example by executing the following command:

```
spark-submit --class org.apache.spark.examples.SparkPi \  
--master local \  
$SPARK_HOME/lib/spark-examples*.jar 10
```

6. If the installation was successful, you should see something similar to the following result (omitting the informational log messages). Note, this is an estimator program, so the actual result may vary:

```
Pi is roughly 3.140576
```



## TRY IT YOURSELF

### Install Spark on Microsoft Windows

Installing Spark on Windows can be more involved than installing it on Linux or Mac OS X because many of the dependencies (such as Python and Java) need to be addressed first.

This example uses a Windows Server 2012, the server version of Windows 8.

1. You will need a decompression utility capable of extracting `.tar.gz` and `.gz` archives because Windows does not have native support for these archives. 7-zip is a suitable program for this. You can obtain it from <http://7-zip.org/download.html>.
2. As shown in Figure 3.1, download the `spark-1.5.2-bin-hadoop2.6.tgz` package from your local mirror and extract the contents of this archive to a new directory called `C:\Spark`.
3. Install Java using the Oracle JDK Version 1.7, which you can obtain from the Oracle website. In this example, I download and install the `jdk-7u79-windows-x64.exe` package.
4. Disable IPv6 for Java applications by running the following command as an administrator from the Windows command prompt :

```
setx /M _JAVA_OPTIONS "-Djava.net.preferIPv4Stack=true"
```

5. Python is not included with Windows, so you will need to download and install it. You can obtain a Windows installer for Python from <https://www.python.org/getit/>. I use Python 2.7.10 in this example. Install Python into `C:\Python27`.
6. Download the Hadoop common binaries necessary to run Spark compiled for Windows x64 from `hadoop-common-bin`. Extract these files to a new directory called `C:\Hadoop`.

7. Set an environment variable at the machine level for `HADOOP_HOME` by running the following command as an administrator from the Windows command prompt:

```
setx /M HADOOP_HOME C:\Hadoop
```

8. Update the system path by running the following command as an administrator from the Windows command prompt:

```
setx /M path "%path%;C:\Python27;%PROGRAMFILES%\Java\jdk1.7.0_79\bin;C:\Hadoop"
```


9. Make a temporary directory, `C:\tmp\hive`, to enable the `HiveContext` in Spark. Set permission to this file using the `winutils.exe` program included with the Hadoop common binaries by running the following commands as an administrator from the Windows command prompt:

```
mkdir C:\tmp\hive
C:\Hadoop\bin\winutils.exe chmod 777 /tmp/hive
```

10. Test the Spark interactive shell in Python by running the following command:

```
C:\Spark\bin\pyspark
```

You should see the output shown in Figure 3.2.

```
Administrator: C:\Windows\system32\cmd.exe - c:\Spark\bin\pyspark
port 49628.
15/11/29 09:36:56 INFO SparkEnv: Registering OutputCommitCoordinator
15/11/29 09:36:56 INFO Utils: Successfully started service 'SparkUI' on port 4040.
15/11/29 09:36:56 INFO SparkUI: Started SparkUI at http://172.31.9.69:4040
15/11/29 09:36:56 WARN MetricsSystem: Using default name DAGScheduler for source
because spark.app.id is not set.
15/11/29 09:36:56 INFO Executor: Starting executor ID driver on host localhost
15/11/29 09:36:56 INFO Utils: Successfully started service 'org.apache.spark.net
work.netty.NettyBlockTransferService' on port 49635.
15/11/29 09:36:56 INFO NettyBlockTransferService: Server created on 49635
15/11/29 09:36:56 INFO BlockManagerMaster: Trying to register BlockManager
15/11/29 09:36:56 INFO BlockManagerMasterEndpoint: Registering block manager loc
alhost:49635 with 534.5 MB RAM, BlockManagerId(driver, localhost, 49635)
15/11/29 09:36:56 INFO BlockManagerMaster: Registered BlockManager
Welcome to
 version 1.5.2
Using Python version 2.7.6 (default, Nov 10 2013 19:24:24)
SparkContext available as sc, HiveContext available as sqlContext.
>>>
```

**FIGURE 3.2**

The PySpark shell in Windows.

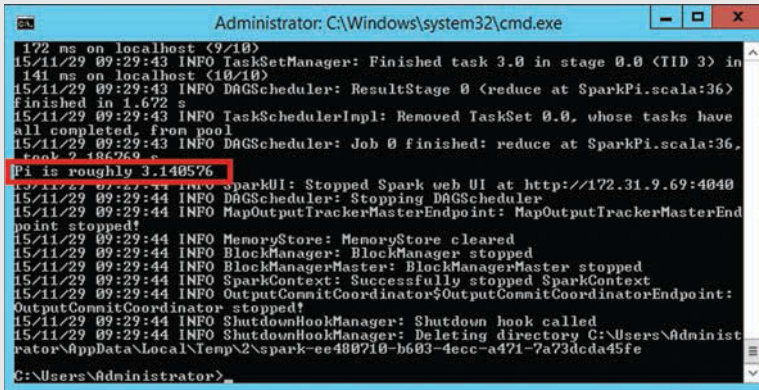
11. You should get a similar result by running the following command to open an interactive Scala shell:

```
C:\Spark\bin\spark-shell
```

12. Run the included Pi Estimator example by executing the following command:

```
C:\Spark\bin\spark-submit --class org.apache.spark.examples.SparkPi --master
local C:\Spark\lib\spark-examples*.jar 10
```

13. If the installation was successful, you should see something similar to the following result shown in Figure 3.3. Note, this is an estimator program, so the actual result may vary:



```
Administrator: C:\Windows\system32\cmd.exe
172 ms on localhost (9/10)
15/11/29 09:29:43 INFO TaskSetManager: Finished task 3.0 in stage 0.0 (TID 3) in
141 ms on localhost (10/10)
15/11/29 09:29:43 INFO DAGScheduler: ResultStage 0 (reduce at SparkPi.scala:36)
finished in 1.672 s
15/11/29 09:29:43 INFO TaskSchedulerImpl: Removed TaskSet 0.0, whose tasks have
all completed, from pool
15/11/29 09:29:43 INFO DAGScheduler: Job 0 finished: reduce at SparkPi.scala:36,
rdd = 2 + 406260
Pi is roughly 3.140576
15/11/29 09:29:44 INFO sparkUI: Stopped Spark web UI at http://172.31.9.69:4040
15/11/29 09:29:44 INFO DAGScheduler: Stopping DAGScheduler
15/11/29 09:29:44 INFO MapOutputTrackerMasterEndpoint: MapOutputTrackerMasterEnd
point stopped!
15/11/29 09:29:44 INFO MemoryStore: MemoryStore cleared
15/11/29 09:29:44 INFO BlockManager: BlockManager stopped
15/11/29 09:29:44 INFO BlockManagerMaster: BlockManagerMaster stopped
15/11/29 09:29:44 INFO SparkContext: Successfully stopped SparkContext
15/11/29 09:29:44 INFO OutputCommitCoordinator$OutputCommitCoordinatorEndpoint:
OutputCommitCoordinator stopped!
15/11/29 09:29:44 INFO ShutdownHookManager: Shutdown hook called
15/11/29 09:29:44 INFO ShutdownHookManager: Deleting directory C:\Users\Administ
rator\AppData\Local\Temp\2\spark-ee480710-b603-4ecc-a471-7a73dca45fe
C:\Users\Administrator>
```

**FIGURE 3.3**

The results of the SparkPi example program in Windows.

## Installing a Multi-node Spark Standalone Cluster

Using the steps outlined in this section for your preferred target platform, you will have installed a single node Spark Standalone cluster. I will discuss Spark's cluster architecture in more detail in Hour 4, "Understanding the Spark Runtime Architecture." However, to create a multi-node cluster from a single node system, you would need to do the following:

- ▶ Ensure all cluster nodes can resolve hostnames of other cluster members and are routable to one another (typically, nodes are on the same private subnet).
- ▶ Enable passwordless SSH (Secure Shell) for the Spark master to the Spark slaves (this step is only required to enable remote login for the slave daemon startup and shutdown actions).
- ▶ Configure the `spark-defaults.conf` file on all nodes with the URL of the Spark master node.
- ▶ Configure the `spark-env.sh` file on all nodes with the hostname or IP address of the Spark master node.
- ▶ Run the `start-master.sh` script from the `sbin` directory on the Spark master node.
- ▶ Run the `start-slave.sh` script from the `sbin` directory on all of the Spark slave nodes.
- ▶ Check the Spark master UI. You should see each slave node in the `Workers` section.
- ▶ Run a test Spark job.

## TRY IT YOURSELF ▼

## Configuring and Testing a Multinode Spark Cluster

Take your single node Spark system and create a basic two-node Spark cluster with a master node and a worker node.

In this example, I use two Linux instances with Spark installed in the same relative paths: one with a hostname of `sparkmaster`, and the other with a hostname of `sparkslave`.

1. Ensure that each node can resolve the other. The `ping` command can be used for this. For example, from `sparkmaster`:
2. Ensure the firewall rules of network ACLs will allow traffic on multiple ports between cluster instances because cluster nodes will communicate using various TCP ports (normally not a concern if all cluster nodes are on the same subnet).
3. Create and configure the `spark-defaults.conf` file on all nodes. Run the following commands on the `sparkmaster` and `sparkslave` hosts:

```
cd $SPARK_HOME/conf
sudo cp spark-defaults.conf.template spark-defaults.conf
sudo sed -i "\$aspark.master\tspark://sparkmaster:7077" spark-defaults.conf
```

4. Create and configure the `spark-env.sh` file on all nodes. Complete the following tasks on the `sparkmaster` and `sparkslave` hosts:

```
cd $SPARK_HOME/conf
sudo cp spark-env.sh.template spark-env.sh
sudo sed -i "\$aSPARK_MASTER_IP=sparkmaster" spark-env.sh
```

5. On the `sparkmaster` host, run the following command:

```
sudo $SPARK_HOME/sbin/start-master.sh
```

6. On the `sparkslave` host, run the following command:

```
sudo $SPARK_HOME/sbin/start-slave.sh spark://sparkmaster:7077
```

7. Check the Spark master web user interface (UI) at <http://sparkmaster:8080/>.

8. Check the Spark worker web UI at <http://sparkslave:8081/>.

9. Run the built-in Pi Estimator example from the terminal of either node:

```
spark-submit --class org.apache.spark.examples.SparkPi \
--master spark://sparkmaster:7077 \
--driver-memory 512m \
--executor-memory 512m \
--executor-cores 1 \
$SPARK_HOME/lib/spark-examples*.jar 10
```



10. If the application completes successfully, you should see something like the following (omitting informational log messages). Note, this is an estimator program, so the actual result may vary:

```
Pi is roughly 3.140576
```

This is a simple example. If it was a production cluster, I would set up passwordless SSH to enable the `start-all.sh` and `stop-all.sh` shell scripts. I would also consider modifying additional configuration parameters for optimization.

## CAUTION

### Spark Master Is a Single Point of Failure in Standalone Mode

Without implementing *High Availability (HA)*, the Spark Master node is a *single point of failure (SPOF)* for the Spark cluster. This means that if the Spark Master node goes down, the Spark cluster would stop functioning, all currently submitted or running applications would fail, and no new applications could be submitted.

High Availability can be configured using *Apache Zookeeper*, a highly reliable distributed coordination service. You can also configure HA using the filesystem instead of Zookeeper; however, this is not recommended for production systems.

## Exploring the Spark Install

Now that you have Spark up and running, let's take a closer look at the install and its various components.

If you followed the instructions in the previous section, "Installing Spark in Standalone Mode," you should be able to browse the contents of `$SPARK_HOME`.

In Table 3.1, I describe each subdirectory of the Spark installation.

**TABLE 3.1** Spark Installation Subdirectories

Directory	Description
bin	Contains all of the commands/scripts to run Spark applications interactively through shell programs such as <code>pyspark</code> , <code>spark-shell</code> , <code>spark-sql</code> and <code>sparkR</code> , or in batch mode using <code>spark-submit</code> .
conf	Contains templates for Spark configuration files, which can be used to set Spark environment variables ( <code>spark-env.sh</code> ) or set default master, slave, or client configuration parameters ( <code>spark-defaults.conf</code> ). There are also configuration templates to control logging ( <code>log4j.properties</code> ), metrics collection ( <code>metrics.properties</code> ), as well as a template for the <code>slaves</code> file, which controls which slave nodes can join the Spark cluster.

Directory	Description
ec2	Contains scripts to deploy Spark nodes and clusters on Amazon Web Services (AWS) Elastic Compute Cloud (EC2). I will cover deploying Spark in EC2 in <b>Hour 5, “Deploying Spark in the Cloud.”</b>
lib	Contains the main assemblies for Spark including the main library ( <code>spark-assembly-x.x.x-hadoopx.x.x.jar</code> ) and included example programs ( <code>spark-examples-x.x.x-hadoopx.x.x.jar</code> ), of which we have already run one, <code>SparkPi</code> , to verify the installation in the previous section.
licenses	Includes license files covering other included projects such as Scala and JQuery. These files are for legal compliance purposes only and are not required to run Spark.
python	Contains all of the Python libraries required to run PySpark. You will generally not need to access these files directly.
sbin	Contains administrative scripts to start and stop master and slave services (locally or remotely) as well as start processes related to YARN and Mesos. I used the <code>start-master.sh</code> and <code>start-slave.sh</code> scripts when I covered how to install a multi-node cluster in the previous section.
data	Contains sample data sets used for testing <code>mllib</code> (which we will discuss in more detail in <b>Hour 16, “Machine Learning with Spark”</b> ).
examples	Contains the source code for all of the examples included in <code>lib/spark-examples-x.x.x-hadoopx.x.x.jar</code> . Example programs are included in Java, Python, R, and Scala. You can also find the latest code for the included examples at <a href="https://github.com/apache/spark/tree/master/examples">https://github.com/apache/spark/tree/master/examples</a> .
R	Contains the <code>SparkR</code> package and associated libraries and documentation. I will discuss <code>SparkR</code> in <b>Hour 15, “Getting Started with Spark and R”</b>

## Deploying Spark on Hadoop

As discussed previously, deploying Spark with Hadoop is a popular option for many users because Spark can read from and write to the data in Hadoop (in HDFS) and can leverage Hadoop’s process scheduling subsystem, YARN.

### Using a Management Console or Interface

If you are using a commercial distribution of Hadoop such as Cloudera or Hortonworks, you can often deploy Spark using the management console provided with each respective platform: for example, Cloudera Manager for Cloudera or Ambari for Hortonworks.



If you are using the management facilities of a commercial distribution, the version of Spark deployed may lag the latest stable Apache release because Hadoop vendors typically update their software stacks with their respective major and minor release schedules.

## Installing Manually

Installing Spark on a YARN cluster manually (that is, not using a management interface such as Cloudera Manager or Ambari) is quite straightforward to do.

### ▼ TRY IT YOURSELF

#### Installing Spark on Hadoop Manually

1. Follow the steps outlined for your target platform (for example, Red Hat Linux, Windows, and so on) in the earlier section “Installing Spark in Standalone Mode.”
2. Ensure that the system you are installing on is a Hadoop client with configuration files pointing to a Hadoop cluster. You can do this as shown:

```
hadoop fs -ls
```

This lists the contents of your user directory in HDFS. You could instead use the path in HDFS where your input data resides, such as

```
hadoop fs -ls /path/to/my/data
```

If you see an error such as `hadoop: command not found`, you need to make sure a correctly configured Hadoop client is installed on the system before continuing.

3. Set either the `HADOOP_CONF_DIR` or `YARN_CONF_DIR` environment variable as shown:

```
export HADOOP_CONF_DIR=/etc/hadoop/conf
# or
export YARN_CONF_DIR=/etc/hadoop/conf
```

As with `SPARK_HOME`, these variables could be set using the `.bashrc` or similar profile script sourced automatically.

4. Execute the following command to test Spark on YARN:

```
spark-submit --class org.apache.spark.examples.SparkPi \
--master yarn-cluster \
$SPARK_HOME/lib/spark-examples*.jar 10
```

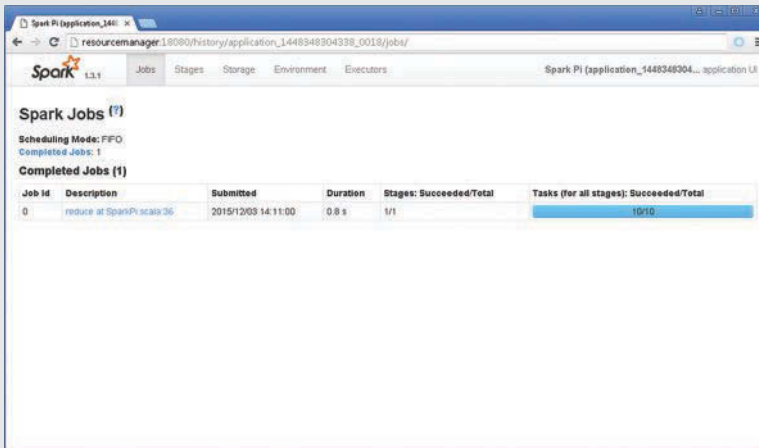
5. If you have access to the YARN Resource Manager UI, you can see the Spark job running in YARN as shown in Figure 3.4:



**FIGURE 3.4**

The YARN ResourceManager UI showing the Spark application running.

6. Clicking the **ApplicationsMaster** link in the **ResourceManager** UI will redirect you to the Spark UI for the application:



**FIGURE 3.5**

The Spark UI.

Submitting Spark applications using YARN can be done in two submission modes: `yarn-cluster` or `yarn-client`.

Using the `yarn-cluster` option, the Spark Driver and Spark Context, ApplicationsMaster, and all executors run on YARN NodeManagers. These are all concepts we will explore in detail in **Hour 4, “Understanding the Spark Runtime Architecture.”** The `yarn-cluster` submission mode is intended for production or non interactive/batch Spark applications. You cannot use

`yarn-cluster` as an option for any of the interactive Spark shells. For instance, running the following command:

```
spark-shell --master yarn-cluster
```

will result in this error:

```
Error: Cluster deploy mode is not applicable to Spark shells.
```

Using the `yarn-client` option, the Spark Driver runs on the client (the host where you ran the Spark application). All of the tasks and the ApplicationsMaster run on the YARN NodeManagers however unlike `yarn-cluster` mode, the Driver does not run on the ApplicationsMaster. The `yarn-client` submission mode is intended to run interactive applications such as `pyspark` or `spark-shell`.

## CAUTION

---

### Running Incompatible Workloads Alongside Spark May Cause Issues

Spark is a memory-intensive processing engine. Using Spark on YARN will allocate containers, associated CPU, and memory resources to applications such as Spark as required. If you have other memory-intensive workloads, such as Impala, Presto, or HAWQ running on the cluster, you need to ensure that these workloads can coexist with Spark and that neither compromises the other. Generally, this can be accomplished through application, YARN cluster, scheduler, or application queue configuration and, in extreme cases, operating system cgroups (on Linux, for instance).

---

## Summary

In this hour, I have covered the different deployment modes for Spark: Spark Standalone, Spark on Mesos, and Spark on YARN.

Spark Standalone refers to the built-in process scheduler it uses as opposed to using a preexisting external scheduler such as Mesos or YARN. A Spark Standalone cluster could have any number of nodes, so the term “Standalone” could be a misnomer if taken out of context. I have showed you how to install Spark both in Standalone mode (as a single node or multi-node cluster) and how to install Spark on an existing YARN (Hadoop) cluster.

I have also explored the components included with Spark, many of which you will have used by the end of this book.

You’re now up and running with Spark. You can use your Spark installation for most of the exercises throughout this book.

## Q&A

**Q. What are the factors involved in selecting a specific deployment mode for Spark?**

- A.** The choice of deployment mode for Spark is primarily dependent upon the environment you are running in and the availability of external scheduling frameworks such as YARN or Mesos. For instance, if you are using Spark with Hadoop and you have an existing YARN infrastructure, Spark on YARN is a logical deployment choice. However, if you are running Spark independent of Hadoop (for instance sourcing data from S3 or a local filesystem), Spark Standalone may be a better deployment method.

**Q. What is the difference between the `yarn-client` and the `yarn-cluster` options of the `--master` argument using `spark-submit`?**

- A.** Both the `yarn-client` and `yarn-cluster` options execute the program in the Hadoop cluster using YARN as the scheduler; however, the `yarn-client` option uses the client host as the driver for the program and is designed for testing as well as interactive shell usage.

## Workshop

The workshop contains quiz questions and exercises to help you solidify your understanding of the material covered. Try to answer all questions before looking at the “Answers” section that follows.

### Quiz

- 1. True or false:** A Spark Standalone cluster consists of a single node.
- 2.** Which component is not a prerequisite for installing Spark?
  - A.** Scala
  - B.** Python
  - C.** Java
- 3.** Which of the following subdirectories contained in the Spark installation contains scripts to start and stop master and slave node Spark services?
  - A.** `bin`
  - B.** `sbin`
  - C.** `lib`
- 4.** Which of the following environment variables are required to run Spark on Hadoop/YARN?
  - A.** `HADOOP_CONF_DIR`
  - B.** `YARN_CONF_DIR`
  - C.** Either `HADOOP_CONF_DIR` or `YARN_CONF_DIR` will work.

## Answers

1. **False.** Standalone refers to the independent process scheduler for Spark, which could be deployed on a cluster of one-to-many nodes.
2. **A.** The Scala assembly is included with Spark; however, Java and Python must exist on the system prior to installation.
3. **B.** `sbin` contains administrative scripts to start and stop Spark services.
4. **C.** Either the `HADOOP_CONF_DIR` or `YARN_CONF_DIR` environment variable must be set for Spark to use YARN.

## Exercises

1. Using your Spark Standalone installation, execute `pyspark` to open a PySpark interactive shell.
2. Open a browser and navigate to the SparkUI at <http://localhost:4040>.
3. Click the Environment top menu link or navigate to Environment page directly using the url: <http://localhost:4040/environment/>.
4. Note some of the various environment settings and configuration parameters set. I will explain many of these in greater detail throughout the book.

# Index

## Symbols

`<-` (assignment operator) in R, 344

## A

ABC programming language, 166

abstraction, Spark as, 2

access control lists (ACLs), 503

accumulator() method, 266

accumulators, 265–266

    accumulator() method, 266

    custom accumulators, 267

    in DStreams, 331, 340

    usage example, 268–270

    value() method, 266

    warning about, 268

ACLs (access control lists), 503

actions

    aggregate actions, 209

        fold(), 210

        reduce(), 209

    collect(), 207

    count(), 206

    defined, 47, 206

    first(), 208–209

    foreach(), 210–211

        map() transformation

        versus, 233

    lazy evaluation, 107–108

    on RDDs, 92

    saveAsHadoopFile(), 251–252

    saveAsNewAPIHadoopFile(),  
        253

    saveAsSequenceFile(), 250

    saveAsTextFile(), 93, 248

    spark-ec2 shell script, 65

    take(), 207–208

    takeSample(), 199

    top(), 208

adjacency lists, 400–401

adjacency matrix, 401–402

aggregation, 209

    fold() method, 210

    foldByKey() method, 217

    groupByKey() method, 202,  
        313–314

    groupByKey() method,  
        215–216, 233

    reduce() method, 209

- reduceByKey() method, 216–217, 233
- sortByKey() method, 217–218
- subtractByKey() method, 218–219
- Alluxio, 254, 258**
  - architecture, 254–255
  - benefits of, 257
  - explained, 254
  - as filesystem, 255–256
  - off-heap persistence, 256
- ALS (Alternating Least Squares), 373**
- Amazon DynamoDB, 429–430**
- Amazon Kinesis Streams. See Kinesis Streams**
- Amazon Machine Image (AMI), 66**
- Amazon Software License (ASL), 448**
- Amazon Web Services (AWS), 61–62**
  - EC2 (Elastic Compute Cloud), 62–63
    - Spark deployment on, 64–73
  - EMR (Elastic MapReduce), 63–64
    - Spark deployment on, 73–80
  - pricing, 64
  - S3 (Simple Storage Service), 63
- AMI (Amazon Machine Image), 66**
- anonymous functions**
  - in Python, 179–180
  - in Scala, 158
- Apache Cassandra. See Cassandra**
- Apache Drill, 290**
- Apache HAWQ, 290**
- Apache Hive. See Hive**
- Apache Kafka. See Kafka**
- Apache Mahout, 367**
- Apache Parquet, 299**
- Apache Software Foundation (ASF), 1**
- Apache Solr, 430**
- Apache Spark. See Spark**
- Apache Storm, 323**
- Apache Tez, 289**
- Apache Zeppelin, 75**
- Apache Zookeeper, 38, 436**
  - installing, 441
- API access to Spark History Server, 489–490**
- appenders in Log4j framework, 493, 499**
- application support in Spark, 3**
- application UI, 48, 479**
  - diagnosing performance problems, 536–539
  - Environment tab, 486
  - example Spark routine, 480
  - Executors tab, 486–487
  - Jobs tab, 481–482
  - in local mode, 57
  - security via Java Servlet Filters, 510–512, 517
  - in Spark History Server, 488–489
  - Stages tab, 483–484
  - Storage tab, 484–485
  - tabs in, 499
- applications**
  - components of, 45–46
    - cluster managers, 49, 51
    - drivers, 46–48
    - executors, 48–49
    - masters, 49–50
    - workers, 48–49
  - defined, 21
  - deployment environment variables, 457
  - external applications
    - accessing Spark SQL, 319
    - processing RDDs with, 278–279
  - managing
    - in Standalone mode, 466–469
    - on YARN, 473–475
  - Map-only applications, 124–125
  - optimizing
    - associative operations, 527–529
    - collecting data, 530
    - diagnosing problems, 536–539
    - dynamic allocation, 531–532
    - with filtering, 527
    - functions and closures, 529–530
    - serialization, 531
  - planning, 47
  - returning results, 48
  - running in local mode, 56–58
  - running on YARN, 20–22, 51, 472–473

- application management, 473–475
- ApplicationsMaster, 52–53
- log file management, 56
- ResourceManager, 51–52, 471–472
- yarn-client submission mode, 54–55
- yarn-cluster submission mode, 53–54
- Scala
  - compiling, 140–141
  - packaging, 141
  - scheduling, 47
    - in Standalone mode, 469–471
    - on YARN, 475–476
  - setting logging within, 497–498
  - viewing status of all, 487
- ApplicationsMaster, 20–21, 471–472
  - as Spark master, 52–53
- arrays in R, 345
- ASF (Apache Software Foundation), 1
- ASL (Amazon Software License), 448
- assignment operator (<-) in R, 344
- associative operations, 209
  - optimizing, 527–529
- asymmetry, speculative execution and, 124
- attribute value pairs. *See* key value pairs (KVP)
- authentication, 503–504
  - encryption, 506–510
  - with Java Servlet Filters, 510–511

- with Kerberos, 512–514, 517
  - client commands, 514
  - configuring, 515–516
  - with Hadoop, 514–515
  - terminology, 513
  - shared secrets, 504–506
- authentication service (AS), 513
- authorization, 503–504
  - with Java Servlet Filters, 511–512
- AWS (Amazon Web Services).  
*See* Amazon Web Services (AWS)

## B

- BackType, 323
- Bagel, 403
- Bayes' Theorem, 372
- Beeline, 287, 318–321
- Beeswax, 287
- benchmarks, 519–520
  - spark-perf, 521–525
  - Terasort, 520–521
  - TPC (Transaction Processing Performance Council), 520
  - when to use, 540
- big data, history of, 11–12
- Bigtable, 417–418
- bin directory, 38
- block reports, 17
- blocks
  - in HDFS, 14–16
  - replication, 25
- bloom filters, 422
- bound variables, 158

- breaking for loops, 151
- broadcast() method, 260–261
- broadcast variables, 259–260
  - advantages of, 263–265, 280
  - broadcast() method, 260–261
  - configuration options, 262
  - in DStreams, 331
  - unpersist() method, 262
  - usage example, 268–270
  - value() method, 261–262
- brokers in Kafka, 436
- buckets, 63
- buffering messages, 435
- built-in functions for DataFrames, 310
- bytecode, machine code versus, 168

## C

- c() method (combine), 346
- cache() method, 108, 314
- cacheTable() method, 314
- caching
  - DataFrames, 314
  - DStreams, 331
  - RDDs, 108–109, 239–240, 243
- callback functions, 180
- canary queries, 525
- CapacityScheduler, 52
- capitalization. *See* naming conventions
- cartesian() method, 225–226
- case statement in Scala, 152



**Cassandra**

- accessing via Spark, 427–429
- CQL (Cassandra Query Language), 426–427
- data model, 426
- HBase versus, 425–426, 431
- Cassandra Query Language (CQL), 426–427**
- Centos, installing Spark, 30–31**
- centroids in clustering, 366
- character data type in R, 345
- character functions in R, 349
- checkpoint() method, 244–245
- checkpointing
  - defined, 111
  - DStreams, 330–331, 340
  - RDDs, 244–247, 258
- checksums, 17
- child RDDs, 109
- choosing. *See* selecting
- classes in Scala, 153–155
- classification in machine learning, 364, 367
  - decision trees, 368–372
  - Naive Bayes, 372–373
- clearCache() method, 314
- CLI (command line interface) for Hive, 287**
- clients
  - in Kinesis Streams, 448
  - MQTT, 445
- closures
  - optimizing applications, 529–530
  - in Python, 181–183
  - in Scala, 158–159

**cloud deployment**

- on Databricks, 81–88
- on EC2, 64–73
- on EMR, 73–80
- Cloudera Impala, 289**
- cluster architecture in Kafka, 436–437
- cluster managers, 45, 49, 51
  - independent variables, 454–455
  - ResourceManager as, 51–52
- cluster mode (EMR), 74
- clustering in machine learning, 365–366, 375–377
- clustering keys in Cassandra, 426
- clusters
  - application deployment
    - environment variables, 457
  - defined, 13
  - EMR launch modes, 74
  - master UI, 487
  - operational overview, 22–23
  - Spark Standalone mode. *See* Spark Standalone deployment mode
- coalesce() method, 274–275, 314**
- coarse-grained transformations, 107
- codecs, 94, 249
- cogroup() method, 224–225
- CoGroupedRDDs, 112**
- collaborative filtering in machine learning, 365, 373–375
- collect() method, 207, 306, 530
- collections
  - in Cassandra, 426
  - diagnosing performance problems, 538–539

## in Scala, 144

- lists, 145–146, 163
- maps, 148–149
- sets, 146–147, 163
- tuples, 147–148
- column families, 420**
- columnar storage formats, 253, 299
- columns method, 305
- Combiner functions, 122–123**
- command line interface (CLI) for Hive, 287
- commands, spark-submit, 7, 8
- committers, 2
- commutative operations, 209
- comparing objects in Scala, 143
- compiling Scala programs, 140–141
- complex data types in Spark SQL, 302
- components (in R vectors), 345
- compression
  - external storage, 249–250
  - of files, 93–94
  - Parquet files, 300
- conf directory, 38**
- configuring
  - Kerberos, 515–516
  - local mode options, 56–57
  - Log4j framework, 493–495
  - SASL, 509
  - Spark
    - broadcast variables, 262
    - configuration properties, 457–460, 477
    - environment variables, 454–457

- managing configuration, 461
- precedence, 460–461
- Spark History Server, 488
- SSL, 506–510
- connected components algorithm, 405**
- consumers**
  - defined, 434
  - in Kafka, 435
- containers, 20–21**
- content filtering, 434–435, 451**
- contributors, 2**
- control structures in Scala, 149**
  - do while and while loops, 151–152
  - for loops, 150–151
  - if expressions, 149–150
  - named functions, 153
  - pattern matching, 152
- converting DataFrames to RDDs, 301**
- core nodes, task nodes versus, 89**
- Couchbase, 430**
- CouchDB, 430**
- count() method, 206, 306**
- counting words. See Word Count algorithm (MapReduce example)**
- cPickle, 176**
- CPython, 167–169**
- CQL (Cassandra Query Language), 426–427**
- CRAN packages in R, 349**
- createDataFrame() method, 294–295**
- createDirectStream() method, 439–440**

- createStream() method**
  - KafkaUtils package, 440
  - KinesisUtils package, 449–450
  - MQTTUtils package, 445–446
- CSV files, creating SparkR data frames from, 352–354**
- current directory in Hadoop, 18**
- Curry, Haskell, 159**
- currying in Scala, 159**
- custom accumulators, 267**
- Cutting, Doug, 11–12, 115**

## D

- daemon logging, 495**
- DAG (directed acyclic graph), 47, 399**
- Data Definition Language (DDL) in Hive, 288**
- data deluge**
  - defined, 12
  - origin of, 117
- data directory, 39**
- data distribution in HBase, 422**
- data frames**
  - matrices versus, 361
  - in R, 345, 347–348
  - in SparkR
    - creating from CSV files, 352–354
    - creating from Hive tables, 354–355
    - creating from R data frames, 351–352

- data locality**
  - defined, 12, 25
  - in loading data, 113
  - with RDDs, 94–95
- data mining, 355. See also R programming language**
- data model**
  - for Cassandra, 426
  - for DataFrames, 301–302
  - for DynamoDB, 429
  - for HBase, 420–422
- data sampling, 198–199**
  - sample() method, 198–199
  - takeSample() method, 199
- data sources**
  - creating
    - JDBC datasources, 100–103
    - relational databases, 100
  - for DStreams, 327–328
  - HDFS as, 24
- data structures**
  - in Python
    - dictionaries, 173–174
    - lists, 170, 194
    - sets, 170–171
    - tuples, 171–173, 194
  - in R, 345–347
  - in Scala, 144
    - immutability, 160
    - lists, 145–146, 163
    - maps, 148–149
    - sets, 146–147, 163
    - tuples, 147–148
- data types**
  - in Hive, 287–288
  - in R, 344–345

- in Scala, 142
- in Spark SQL, 301–302
- Databricks, Spark deployment on, 81–88**
- Databricks File System (DBFS), 81**
- Datadog, 525–526**
- `data.frame()` method, 347
- DataFrameReader, creating**
  - DataFrames with, 298–301**
- DataFrames, 102, 111, 294**
  - built-in functions, 310
  - caching, persisting, repartitioning, 314
  - converting to RDDs, 301
  - creating
    - with DataFrameReader, 298–301
    - from Hive tables, 295–296
    - from JSON files, 296–298
    - from RDDs, 294–295
  - data model, 301–302
  - functional operations, 306–310
  - GraphFrames. *See* GraphFrames
  - metadata operations, 305–306
  - saving to external storage, 314–316
  - schemas
    - defining, 304
    - inferring, 302–304
  - set operations, 311–314
  - UDFs (user-defined functions), 310–311
- DataNodes, 17**
- Dataset API, 118**
- datasets, defined, 92, 117. *See also* RDDs (Resilient Distributed Datasets)
- datasets package, 351–352
- DataStax, 425**
- DBFS (Databricks File System), 81**
- `dbutils.fs`, 89
- DDL (Data Definition Language) in Hive, 288**
- Debian Linux, installing Spark, 32–33**
- decision trees, 368–372**
- DecisionTree.trainClassifier function, 371–372**
- deep learning, 381–382**
- defaults for environment variables and configuration properties, 460**
- defining DataFrame schemas, 304**
- degrees method, 408–409**
- deleting objects (HDFS), 19**
- deploying. *See also* installing**
  - cluster applications, environment variables for, 457
  - H2O on Hadoop, 384–386
  - Spark
    - on Databricks, 81–88
    - on EC2, 64–73
    - on EMR, 73–80
    - Spark History Server, 488
- deployment modes for Spark. *See also* Spark on YARN**
  - deployment mode; Spark**
  - Standalone deployment mode**
    - list of, 27–28
    - selecting, 43
- describe method, 392**
- design goals for MapReduce, 117**
- destructuring binds in Scala, 152**
- diagnosing performance problems, 536–539**
- dictionaries**
  - `keys()` method, 212
  - in Python, 101, 173–174
  - `values()` method, 212
- direct stream access in Kafka, 438, 451**
- directed acyclic graph (DAG), 47, 399**
- directory contents**
  - listing, 19
  - subdirectories of Spark installation, 38–39
- discretized streams. *See* DStreams**
- distinct() method, 203–204, 308**
- distributed, defined, 92**
- distributed systems, limitations of, 115–116**
- distribution of blocks, 15**
- do while loops in Scala, 151–152**
- docstrings, 310**
- document stores, 419**
- documentation for Spark SQL, 310**
- DoubleRDDs, 111**
- downloading**
  - files, 18–19
  - Spark, 29–30
- Drill, 290**
- drivers, 45, 46–48**
  - application planning, 47
  - application scheduling, 47
  - application UI, 48
  - masters versus, 50

returning results, 48

SparkContext, 46–47

**drop() method, 307**

**DStream.checkpoint() method, 330**

**DStreams (discretized streams), 324, 326–327**

broadcast variables and  
accumulators, 331

caching and persistence, 331

checkpointing, 330–331, 340

data sources, 327–328

lineage, 330

output operations, 331–333

sliding window operations,  
337–339, 340

state operations, 335–336,  
340

transformations, 328–329

**dtypes method, 305–306**

**Dynamic Resource Allocation,  
476, 531–532**

**DynamoDB, 429–430**

## E

**EBS (Elastic Block Store), 62, 89**

**EC2 (Elastic Compute Cloud),  
62–63, 64–73**

**ec2 directory, 39**

**ecosystem projects, 13**

**edge nodes, 502**

**EdgeRDD objects, 404–405**

**edges**

creating edge DataFrames, 407

in DAG, 47

defined, 399

**edges method, 407–408**

**Elastic Block Store (EBS), 62, 89**

**Elastic Compute Cloud (EC2),  
62–63, 64–73**

**Elastic MapReduce (EMR), 63–64,  
73–80**

**ElasticSearch, 430**

**election analogy for MapReduce,  
125–126**

**encryption, 506–510**

**Environment tab (application UI),  
486, 499**

**environment variables, 454**

cluster application

deployment, 457

cluster manager independent  
variables, 454–455

defaults, 460

Hadoop-related, 455

Spark on YARN environment  
variables, 456–457

Spark Standalone daemon,  
455–456

**ephemeral storage, 62**

**ETags, 63**

**examples directory, 39**

**exchange patterns. See pub-sub  
messaging model**

**executors, 45, 48–49**

logging, 495–497

number of, 477

in Standalone mode, 463

workers versus, 59

**Executors tab (application UI),  
486–487, 499**

**explain() method, 310**

**external applications**

accessing Spark SQL, 319

processing RDDs with,  
278–279

**external storage for RDDs,  
247–248**

Alluxio, 254–257, 258

columnar formats, 253, 299

compressed options, 249–250

Hadoop input/output formats,  
251–253

saveAsTextFile() method, 248

saving DataFrames to,  
314–316

sequence files, 250

**external tables (Hive), internal  
tables versus, 289**

## F

**FairScheduler, 52, 470–471, 477**

**fault tolerance**

in MapReduce, 122

with RDDs, 111

**fault-tolerant mode (Alluxio),  
254–255**

**feature extraction, 366–367, 378**

**features in machine learning,  
366–367**

**files**

compression, 93–94

CSV files, creating SparkR  
data frames from, 352–354

downloading, 18–19

in HDFS, 14–16

JSON files, creating RDDs  
from, 103–105

object files, creating RDDs  
from, 99

text files

creating DataFrames from,  
298–299

- creating RDDs from, 93–99
- saving DStreams as, 332–333
- uploading (ingesting), 18
- filesystem, Alluxio as, 255–256**
- filter() method, 201–202, 307**
  - in Python, 170
- filtering**
  - messages, 434–435, 451
  - optimizing applications, 527
- find method, 409–410**
- fine-grained transformations, 107**
- first() method, 208–209**
- first-class functions in Scala, 157, 163**
- flags for RDD storage levels, 237–238**
- flatMap() method, 131, 200–201**
  - in DataFrames, 308–309
  - map() method versus, 135, 232
- flatMapValues() method, 213–214**
- fold() method, 210**
- foldByKey() method, 217**
- followers in Kafka, 436–437**
- foreach() method, 210–211, 306**
  - map() method versus, 233
- foreachPartition() method, 276–277**
- foreachRDD() method, 333**
- for loops in Scala, 150–151**
- free variables, 158**
- frozensets in Python, 171**
- full outer joins, 219**
- fullOuterJoin() method, 223–224**
- function literals, 163**
- function values, 163**

- functional programming**
  - in Python, 178
    - anonymous functions, 179–180
    - closures, 181–183
    - higher-order functions, 180, 194
    - parallelization, 181
    - short-circuiting, 181
    - tail calls, 180–181
  - in Scala
    - anonymous functions, 158
    - closures, 158–159
    - currying, 159
    - first-class functions, 157, 163
    - function literals versus function values, 163
    - higher-order functions, 158
    - immutable data structures, 160
    - lazy evaluation, 160
- functional transformations, 199**
  - filter() method, 201–202
  - flatMap() method, 200–201
    - map() method versus, 232
  - flatMapValues() method, 213–214
  - keyBy() method, 213
  - map() method, 199–200
    - flatMap() method versus, 232
    - foreach() method versus, 233
  - mapValues() method, 213
- functions**
  - optimizing applications, 529–530

- passing to map
  - transformations, 540–541
- in R, 348–349

**Funnel project, 138**

**future of NoSQL, 430**

## G

- garbage collection, 169**
- gateway services, 503**
- generalized linear model, 357**
- Generic Java (GJ), 137**
- getCheckpointFile() method, 245**
- getStorageLevel() method, 238–239**
- glm() method, 357**
- glom() method, 277**
- Google**
  - graphs and, 402–403
  - in history of big data, 11–12
  - PageRank. *See* PageRank
- graph stores, 419**
- GraphFrames, 406**
  - accessing, 406
  - creating, 407
  - defined, 414
  - methods in, 407–409
  - motifs, 409–410, 414
  - PageRank implementation, 411–413
  - subgraphs, 410
- GraphRDD objects, 405**
- graphs**
  - adjacency lists, 400–401
  - adjacency matrix, 401–402

- characteristics of, 399
  - defined, 399
  - Google and, 402–403
  - GraphFrames, 406
    - accessing, 406
    - creating, 407
    - defined, 414
    - methods in, 407–409
    - motifs, 409–410, 414
    - PageRank implementation, 411–413
    - subgraphs, 410
  - GraphX API, 403–404
    - EdgeRDD objects, 404–405
    - graphing algorithms in, 405
    - GraphRDD objects, 405
    - VertexRDD objects, 404
  - terminology, 399–402
  - GraphX API, 403–404**
    - EdgeRDD objects, 404–405
    - graphing algorithms in, 405
    - GraphRDD objects, 405
    - VertexRDD objects, 404
  - groupBy() method, 202, 313–314**
  - groupByKey() method, 215–216, 233, 527–529**
  - grouping data, 202**
    - distinct() method, 203–204
    - foldByKey() method, 217
    - groupBy() method, 202, 313–314
    - groupByKey() method, 215–216, 233
    - reduceByKey() method, 216–217, 233
    - sortBy() method, 202–203
    - sortByKey() method, 217–218
    - subtractByKey() method, 218–219
- ## H
- H2O, 381**
    - advantages of, 397
    - architecture, 383–384
    - deep learning, 381–382
    - deployment on Hadoop, 384–386
    - interfaces for, 397
    - saving models, 395–396
    - Sparkling Water, 387, 397
      - architecture, 387–388
      - example exercise, 393–395
      - H2OFrames, 390–393
      - pysparkling shell, 388–390
      - web interface for, 382–383
  - H2O Flow, 382–383**
  - H2OContext, 388–390**
  - H2OFrames, 390–393**
  - HA (High Availability), implementing, 38**
  - Hadoop, 115**
    - clusters, 22–23
    - current directory in, 18
    - Elastic MapReduce (EMR), 63–64, 73–80
    - environment variables, 455
    - explained, 12–13
    - external storage, 251–253
    - H2O deployment, 384–386
    - HDFS. *See* HDFS (Hadoop Distributed File System)
    - history of big data, 11–12
    - Kerberos with, 514–515
    - Spark and, 2, 8
      - deploying Spark, 39–42
      - downloading Spark, 30
      - HDFS as data source, 24
      - YARN as resource scheduler, 24
    - SQL on Hadoop, 289–290
    - YARN. *See* YARN (Yet Another Resource Negotiator)
  - Hadoop Distributed File System (HDFS). *See* HDFS (Hadoop Distributed File System)**
  - hadoopFile() method, 99**
  - HadoopRDDs, 111**
  - hash partitioners, 121**
  - Haskell programming language, 159**
  - HAWQ, 290**
  - HBase, 419**
    - Cassandra versus, 425–426, 431
    - data distribution, 422
    - data model and shell, 420–422
    - reading and writing data with Spark, 423–425
  - HCatalog, 286**
  - HDFS (Hadoop Distributed File System), 12**
    - blocks, 14–16
    - DataNodes, 17
    - explained, 13
    - files, 14–16
    - interactions with, 18
      - deleting objects, 19
      - downloading files, 18–19

- listing directory
      - contents, 19
    - uploading (ingesting)
      - files, 18
    - NameNode, 16–17
    - replication, 14–16
    - as Spark data source, 24
  - heap, 49**
  - HFile objects, 422**
  - High Availability (HA), implementing, 38**
  - higher-order functions**
    - in Python, 180, 194
    - in Scala, 158
  - history**
    - of big data, 11–12
    - of IPython, 183–184
    - of MapReduce, 115
    - of NoSQL, 417–418
    - of Python, 166
    - of Scala, 137–138
    - of Spark SQL, 283–284
    - of Spark Streaming, 323–324
  - History Server. *See* Spark History Server**
  - Hive**
    - conventional databases
      - versus, 285–286
    - data types, 287–288
    - DDL (Data Definition Language), 288
    - explained, 284–285
    - interfaces for, 287
    - internal versus external
      - tables, 289
    - metastore, 286
    - Spark SQL and, 291–292
    - tables
      - creating DataFrames from, 295–296
      - creating SparkR data frames from, 354–355
      - writing DataFrame data to, 315
  - Hive on Spark, 284**
  - HiveContext, 292–293, 322**
  - HiveQL, 284–285**
  - HiveServer2, 287**
- I**
- IAM (Identity and Access Management) user accounts, 65**
  - if expressions in Scala, 149–150**
  - immutability**
    - of HDFS, 14
    - of RDDs, 92
  - immutable data structures in Scala, 160**
  - immutable sets in Python, 171**
  - immutable variables in Scala, 144**
  - Impala, 289**
  - indegrees, 400**
  - inDegrees method, 408–409**
  - inferring DataFrame schemas, 302–304**
  - ingesting files, 18**
  - inheritance in Scala, 153–155**
  - initializing RDDs, 93**
    - from datasources, 100
    - from JDBC datasources, 100–103
    - from JSON files, 103–105
    - from object files, 99
    - programmatically, 105–106
    - from text files, 93–99
  - inner joins, 219**
  - input formats**
    - Hadoop, 251–253
    - for machine learning, 371
  - input split, 127**
  - input/output types in Spark, 7**
  - installing. *See also* deploying**
    - IPython, 184–185
    - Jupyter, 189
    - Python, 31
    - R packages, 349
    - Scala, 31, 139–140
    - Spark
      - on Hadoop, 39–42
      - on Mac OS X, 33–34
      - on Microsoft Windows, 34–36
      - as multi-node Standalone cluster, 36–38
      - on Red Hat/Centos, 30–31
      - requirements for, 28
      - in Standalone mode, 29–36
      - subdirectories of
        - installation, 38–39
      - on Ubuntu/Debian Linux, 32–33
      - Zookeeper, 441
  - instance storage, 62**
    - EBS versus, 89
  - Instance Type property (EC2), 62**
  - instances (EC2), 62**
  - int methods in Scala, 143–144**
  - integer data type in R, 345**

**Interactive Computing Protocol, 189**

**Interactive Python. See IPython (Interactive Python)**

**interactive use of Spark, 5–7, 8**

**internal tables (Hive), external tables versus, 289**

**interpreted languages, Python as, 166–167**

**intersect() method, 313**

**intersection() method, 205**

**IoT (Internet of Things)**

defined, 443. *See also* MQTT (MQ Telemetry Transport)

MQTT characteristics for, 451

**IPython (Interactive Python), 183**

history of, 183–184

Jupyter notebooks, 187–189

advantages of, 194

kernels and, 189

with PySpark, 189–193

Spark usage with, 184–187

**IronPython, 169**

**isCheckpointed() method, 245**

## J

**Java, word count in Spark (listing 1.3), 4–5**

**Java Database Connectivity (JDBC) datasources, creating RDDs from, 100–103**

**Java Management Extensions (JMX), 490**

**Java Servlet Filters, 510–512, 517**

**Java virtual machines (JVMs), 139**  
defined, 46

heap, 49

**javac compiler, 137**

**JavaScript Object Notation (JSON). See JSON (JavaScript Object Notation)**

**JDBC (Java Database Connectivity) datasources, creating RDDs from, 100–103**

**JDBC/ODBC interface, accessing Spark SQL, 317–318, 319**

**JdbcRDDs, 112**

**JMX (Java Management Extensions), 490**

**jobs**

in Databricks, 81

diagnosing performance problems, 536–538

scheduling, 470–471

**Jobs tab (application UI), 481–482, 499**

**join() method, 219–221, 312**

**joins, 219**

cartesian() method, 225–226

cogroup() method, 224–225

example usage, 226–229

fullOuterJoin() method, 223–224

join() method, 219–221, 312

leftOuterJoin() method, 221–222

optimizing, 221

rightOuterJoin() method, 222–223

types of, 219

**JSON (JavaScript Object Notation), 174–176**

creating DataFrames from, 296–298

creating RDDs from, 103–105

**json() method, 316**

**jsonFile() method, 104, 297**

**jsonRDD() method, 297–298**

**Jupyter notebooks, 187–189**

advantages of, 194

kernels and, 189

with PySpark, 189–193

**JVMs (Java virtual machines), 139**

defined, 46

heap, 49

**Jython, 169**

## K

**Kafka, 435–436**

cluster architecture, 436–437

Spark support, 437

direct stream access, 438, 451

KafkaUtils package, 439–443

receivers, 437–438, 451

**KafkaUtils package, 439–443**

createDirectStream() method, 439–440

createStream() method, 440

**KCL (Kinesis Client Library), 448**

**KDC (key distribution center), 512–513**



**Kerberos, 512–514, 517**

- client commands, 514
- configuring, 515–516
- with Hadoop, 514–515
- terminology, 513

**kernels, 189****key distribution center (KDC), 512–513****key value pairs (KVP)**

- defined, 118
- in Map phase, 120–121
- pair RDDs, 211
  - flatMapValues() method, 213–214
  - foldByKey() method, 217
  - groupByKey() method, 215–216, 233
  - keyBy() method, 213
  - keys() method, 212
  - mapValues() method, 213
  - reduceByKey() method, 216–217, 233
  - sortByKey() method, 217–218
  - subtractByKey() method, 218–219
  - values() method, 212

**key value stores, 419****keyBy() method, 213****keys, 118****keys() method, 212****keyspaces in Cassandra, 426****keytab files, 513****Kinesis Client Library (KCL), 448****Kinesis Producer Library (KPL), 448****Kinesis Streams, 446–447**

- KCL (Kinesis Client Library), 448
- KPL (Kinesis Producer Library), 448
- Spark support, 448–450

**KinesisUtils package, 448–450****k-means clustering, 375–377****KPL (Kinesis Producer Library), 448****Kryo serialization, 531****KVP (key value pairs). See key value pairs (KVP)****L****LabeledPoint objects, 370****lambda calculus, 119****lambda operator**

- in Java, 5
- in Python, 4, 179–180

**lazy evaluation, 107–108, 160****leaders in Kafka, 436–437****left outer joins, 219****leftOuterJoin() method, 221–222****lib directory, 39****libraries in R, 349****library() method, 349****licenses directory, 39****limit() method, 309****lineage**

- of DStreams, 330
- of RDDs, 109–110, 235–237

**linear regression, 357–358****lines. See edges****linked lists in Scala, 145****Lisp, 119****listing directory contents, 19****listings****accessing**

- Amazon DynamoDB from Spark, 430

- columns in SparkR data frame, 355

- data elements in R matrix, 347

- elements in list, 145

- History Server REST API, 489

- and inspecting data in R data frames, 348

- struct values in motifs, 410

- and using tuples, 148

- Alluxio as off heap memory for RDD persistence, 256

- Alluxio filesystem access using Spark, 256

- anonymous functions in Scala, 158

- appending and prepending to lists, 146

- associative operations in Spark, 527

- basic authentication for Spark UI using Java servlets, 510

- broadcast method, 261

- building generalized linear model with SparkR, 357

- caching RDDs, 240

- cartesian transformation, 226

- Cassandra insert results, 428
- checkpointing
  - RDDs, 245
  - in Spark Streaming, 330
- class and inheritance example
  - in Scala, 154–155
- closures
  - in Python, 182
  - in Scala, 159
- coalesce() method, 275
- cogroup transformation, 225
- collect action, 207
- combine function to create R vector, 346
- configuring
  - pool for Spark application, 471
  - SASL encryption for block transfer services, 509
- connectedComponents
  - algorithm, 405
- converting
  - DataFrame to RDD, 301
  - H2OFrame to Spark SQL DataFrame, 392
- count action, 206
- creating
  - and accessing
    - accumulators, 265
  - broadcast variable from file, 261
  - DataFrame from Hive ORC files, 300
  - DataFrame from JSON document, 297
  - DataFrame from Parquet file (or files), 300
  - DataFrame from plain text file or file(s), 299
  - DataFrame from RDD, 295
  - DataFrame from RDD containing JSON objects, 298
  - edge DataFrame, 407
  - GraphFrame, 407
  - H2OFrame from file, 391
  - H2OFrame from Python object, 390
  - H2OFrame from Spark RDD, 391
  - keyspace and table in Cassandra using cqlsh, 426–427
  - PySparkling H2OContext object, 389
  - R data frame from column vectors, 347
  - R matrix, 347
  - RDD of LabeledPoint objects, 370
  - RDDs from JDBC
    - datasource using load() method, 101
  - RDDs from JDBC
    - datasource using read.jdbc() method, 103
  - RDDs using parallelize() method, 106
  - RDDs using range() method, 106
  - RDDs using textFile() method, 96
  - RDDs using wholeTextFiles() method, 97
  - SparkR data frame from CSV file, 353
  - SparkR data frame from Hive table, 354
  - SparkR data frame from R data frame, 352
  - StreamingContext, 326
  - subgraph, 410
  - table and inserting data in HBase, 420
  - vertex DataFrame, 407
  - and working with RDDs created from JSON files, 104–105
- currying in Scala, 159
- custom accumulators, 267
- declaring lists and using functions, 145
- defining schema
  - for DataFrame explicitly, 304
  - for SparkR data frame, 353
- degrees, inDegrees, and outDegrees methods, 408–409
- detailed H2OFrame
  - information using describe method, 393
- dictionaries in Python, 173–174
- dictionary object usage in PySpark, 174
- dropping columns from DataFrame, 307
- DStream transformations, 329
- EdgeRDDs, 404
- enabling Spark dynamic allocation, 532
- evaluating k-means clustering model, 377

- external transformation
  - program sample, 279
- filtering rows
  - from DataFrame, 307
  - duplicates using distinct, 308
- final output (Map task), 129
- first action, 209
- first five lines of Shakespeare file, 130
- fold action, 210
  - compared with reduce, 210
- foldByKey example to find maximum value by key, 217
- foreach action, 211
- foreachPartition() method, 276
- for loops
  - break, 151
  - with filters, 151
  - in Scala, 150
- fullOuterJoin transformation, 224
- getStorageLevel() method, 239
- getting help for Python API
  - Spark SQL functions, 310
- GLM usage to make prediction on new data, 357
- GraphFrames package, 406
- GraphRDDs, 405
- groupBy transformation, 215
- grouping and aggregating data in DataFrames, 314
- H2OFrame summary function, 392
- higher-order functions
  - in Python, 180
  - in Scala, 158
- Hive CREATE TABLE statement, 288
- human readable
  - representation of Python bytecode, 168–169
- if expressions in Scala, 149–150
- immutable sets in Python and PySpark, 171
- implementing
  - implementing ACLs for Spark UI, 512
  - Naive Bayes classifier using Spark MLlib, 373
- importing graphframe Python module, 406
- including Databricks Spark CSV package in SparkR, 353
- initializing SQLContext, 101
- input to Map task, 127
- int methods, 143–144
- intermediate sent to Reducer, 128
- intersection transformation, 205
- join transformation, 221
- joining DataFrames in Spark SQL, 312
- joining lookup data
  - using broadcast variable, 264
  - using driver variable, 263–264
  - using RDD join(), 263
- JSON object usage
  - in PySpark, 176
  - in Python, 175
- Jupyter notebook JSON document, 188–189
- KafkaUtils.createDirectStream method, 440
- KafkaUtils.createStream (receiver) method, 440
- keyBy transformation, 213
- keys transformation, 212
- Kryo serialization usage, 531
- launching pyspark supplying JDBC MySQL connector JAR file, 101
- lazy evaluation in Scala, 160
- leftOuterJoin transformation, 222
- listing
  - functions in H2O Python module, 389
  - R packages installed and available, 349
- lists
  - with mixed types, 145
  - in Scala, 145
- log events example, 494
- log4j.properties file, 494
- logging events within Spark program, 498
- map, flatMap, and filter transformations in Spark, 201
- map(), reduce(), and filter() in Python and PySpark, 170
- map functions with Spark SQL DataFrames, 309
- mapPartitions() method, 277
- maps in Scala, 148
- mapValues and flatMapValues transformations, 214
- max function, 230
- max values for R integer and numeric (double) types, 345

- mean function, 230
- min function, 230
- mixin composition using traits, 155–156
- motifs, 409–410
- mtcars data frame in R, 352
- mutable and immutable variables in Scala, 144
- mutable maps, 148–149
- mutable sets, 147
- named functions
  - and anonymous functions in Python, 179
  - versus lambda functions in Python, 179
  - in Scala, 153
- non-interactive Spark job submission, 7
- object serialization using Pickle in Python, 176–177
- obtaining application logs from command line, 56
- ordering DataFrame, 313
- output from Map task, 128
- pageRank algorithm, 405
- partitionBy() method, 273
- passing
  - large amounts of data to function, 530
  - Spark configuration properties to spark-submit, 459
- pattern matching in Scala using case, 152
- performing functions in each RDD in DStream, 333
- persisting RDDs, 241–242
- pickleFile() method usage in PySpark, 178
- pipe() method, 279
- PyPy with PySpark, 532
- pyspark command with
  - pyspark-cassandra package, 428
- PySpark interactive shell in local mode, 56
- PySpark program to search for errors in log files, 92
- Python program sample, 168
- RDD usage for multiple actions
  - with persistence, 108
  - without persistence, 108
- reading Cassandra data into Spark RDD, 428
- reduce action, 209
- reduceByKey transformation to average values by key, 216
- reduceByKeyAndWindow function, 339
- repartition() method, 274
- repartitionAndSortWithinPartitions() method, 275
- returning
  - column names and data types from DataFrame, 306
  - list of columns from DataFrame, 305
- rightOuterJoin transformation, 223
- running SQL queries against Spark DataFrames, 102
- sample() usage, 198
- saveAsHadoopFile action, 252
- saveAsNewAPIHadoopFile action, 253
- saveAsPickleFile() method usage in PySpark, 178
- saving
  - DataFrame to Hive table, 315
  - DataFrame to Parquet file or files, 316
  - DStream output to files, 332
  - H2O models in POJO format, 396
  - and loading H2O models in native format, 395
  - RDDs as compressed text files using GZip codec, 249
  - RDDs to sequence files, 250
  - and reloading clustering model, 377
- scanning HBase table, 421
- scheduler XML file example, 470
- schema for DataFrame created from Hive table, 304
- schema inference for DataFrames
  - created from JSON, 303
  - created from RDD, 303
- select method in Spark SQL, 309
- set operations example, 146
- sets in Scala, 146
- setting
  - log levels within application, 497
  - Spark configuration properties programmatically, 458

- spark.scheduler.allocation.  
file property, 471
  - Shakespeare RDD, 130
  - short-circuit operators in  
Python, 181
  - showing current Spark  
configuration, 460
  - simple R vector, 346
  - singleton objects in Scala, 156
  - socketTextStream() method,  
327
  - sortByKey transformation, 218
  - Spark configuration object  
methods, 459
  - Spark configuration properties  
in spark-defaults.conf file,  
458
  - Spark environment variables  
set in spark-env.sh file, 454
  - Spark HiveContext, 293
  - Spark KafkaUtils usage, 439
  - Spark MLlib decision tree  
model to classify new data,  
372
  - Spark pi estimator in local  
mode, 56
  - Spark routine example, 480
  - Spark SQLContext, 292
  - Spark Streaming
    - using Amazon Kinesis,  
449–450
    - using MQTTUtils, 446
  - Spark usage on Kerberized  
Hadoop cluster, 515
  - spark-ec2 syntax, 65
  - spark-perf core tests, 521–522
  - specifying
    - local mode in code, 57
    - log4j.properties file using  
JVM options, 495
  - splitting data into training and  
test data sets, 370
  - sql method for creating  
DataFrame from Hive table,  
295–296
  - state DStreams, 336
  - stats function, 232
  - stdev function, 231
  - StorageClass constructor, 238
  - submitting
    - Spark application to YARN  
cluster, 473
    - streaming application with  
Kinesis support, 448
  - subtract transformation, 206
  - subtractByKey transformation,  
218
  - sum function, 231
  - table method for creating  
dataFrame from Hive table,  
296
  - tail call recursion, 180–181
  - take action, 208
  - takeSample() usage, 199
  - textFileStream() method, 328
  - toDebugString() method, 236
  - top action, 208
  - training
    - decision tree model with  
Spark MLlib, 371
    - k-means clustering model  
using Spark MLlib, 377
  - triangleCount algorithm, 405
  - tuples
    - in PySpark, 173
    - in Python, 172
    - in Scala, 147
  - union transformation, 205
  - unpersist() method, 262
  - updating
    - cells in HBase, 422
    - data in Cassandra table  
using Spark, 428
  - user-defined functions in  
Spark SQL, 311
  - values transformation, 212
  - variance function, 231
  - VertexRDDs, 404
  - vertices and edges methods,  
408
  - viewing applications using  
REST API, 467
  - web log schema sample,  
203–204
  - while and do while loops in  
Scala, 152
  - window function, 338
  - word count in Spark
    - using Java, 4–5
    - using Python, 4
    - using Scala, 4
  - yarn command usage, 475
    - to kill running Spark  
application, 475
  - yield operator, 151
- lists**
- in Python, 170, 194
  - in Scala, 145–146, 163
- load() method, 101–102**
- load\_model function, 395**
- loading data**
- data locality in, 113
  - into RDDs, 93

- from datasources, 100
- from JDBC datasources, 100–103
- from JSON files, 103–105
- from object files, 99
- programmatically, 105–106
- from text files, 93–99
- local mode, running applications, 56–58**
- log aggregation, 56, 497**
- Log4j framework, 492–493**
  - appenders, 493, 499
  - daemon logging, 495
  - executor logs, 495–497
  - log4j.properties file, 493–495
  - severity levels, 493
- log4j.properties file, 493–495**
- loggers, 492**
- logging, 492**
  - Log4j framework, 492–493
    - appenders, 493, 499
    - daemon logging, 495
    - executor logs, 495–497
    - log4j.properties file, 493–495
    - severity levels, 493
  - setting within applications, 497–498
  - in YARN, 56
- logical data type in R, 345**
- logs in Kafka, 436**
- lookup() method, 277**
- loops in Scala**
  - do while and while loops, 151–152
  - for loops, 150–151

## M

**Mac OS X, installing Spark, 33–34**

**machine code, bytecode versus, 168**

**machine learning**

- classification in, 364, 367
  - decision trees, 368–372
  - Naive Bayes, 372–373
- clustering in, 365–366, 375–377
- collaborative filtering in, 365, 373–375
- defined, 363–364
- features and feature extraction, 366–367
- H2O. *See* H2O
- input formats, 371
- in Spark, 367
- Spark MLlib. *See* Spark MLlib
- splitting data sets, 369–370

**Mahout, 367**

**managing**

- applications
  - in Standalone mode, 466–469
  - on YARN, 473–475
- configuration, 461
- performance. *See* performance management

**map() method, 120–121, 130, 199–200**

- in DataFrames, 308–309, 322
- flatMap() method versus, 135, 232
- foreach() method versus, 233
- passing functions to, 540–541

- in Python, 170

- in Word Count algorithm, 129–132

**Map phase, 119, 120–121**

**Map-only applications, 124–125**

**mapPartitions() method, 277–278**

**MapReduce, 115**

- asymmetry and speculative execution, 124
- Combiner functions, 122–123
- design goals, 117
- election analogy, 125–126
- fault tolerance, 122
- history of, 115
- limitations of distributed computing, 115–116
- Map phase, 120–121
- Map-only applications, 124–125
- partitioning function in, 121
- programming model versus processing framework, 118–119
- Reduce phase, 121–122
- Shuffle phase, 121, 135
- Spark versus, 2, 8
- terminology, 117–118
- whitepaper website, 117
- Word Count algorithm
  - example, 126
  - map() and reduce() methods, 129–132
  - operational overview, 127–129
  - in PySpark, 132–134
  - reasons for usage, 126–127
- YARN versus, 19–20

- maps in Scala, 148–149
- mapValues() method, 213
- Marz, Nathan, 323
- master nodes, 23
- master UI, 463–466, 487
- masters, 45, 49–50
  - ApplicationsMaster as, 52–53
  - drivers versus, 50
  - starting in Standalone mode, 463
- match case constructs in Scala, 152
- Mathematica, 183
- matrices
  - data frames versus, 361
  - in R, 345–347
- matrix command, 347
- matrix factorization, 373
- max() method, 230
- MBeans, 490
- McCarthy, John, 119
- mean() method, 230
- members, 111
- Memcached, 430
- memory-intensive workloads,
  - avoiding conflicts, 42
- Mesos, 22
- message oriented middleware (MOM), 433
- messaging systems, 433–434
  - buffering and queueing messages, 435
  - filtering messages, 434–435
  - Kafka, 435–436
    - cluster architecture, 436–437

- direct stream access, 438, 451
  - KafkaUtils package, 439–443
  - receivers, 437–438, 451
  - Spark support, 437
- Kinesis Streams, 446–447
  - KCL (Kinesis Client Library), 448
  - KPL (Kinesis Producer Library), 448
  - Spark support, 448–450
- MQTT, 443
  - characteristics for IoT, 451
  - clients, 445
  - message structure, 445
  - Spark support, 445–446
  - as transport protocol, 444
- pub-sub model, 434–435
- metadata
  - for DataFrames, 305–306
  - in NameNode, 16–17
- metastore (Hive), 286
- metrics, collecting, 490–492
- metrics sinks, 490, 499
- Microsoft Windows, installing Spark, 34–36
- min() method, 229–230
- mixin composition in Scala, 155–156
- MLlib. *See* Spark MLlib
- MOM (message oriented middleware), 433
- MongoDB, 430
- monitoring performance. *See* performance management
- motifs, 409–410, 414

- Movielens dataset, 374
- MQTT (MQ Telemetry Transport), 443
  - characteristics for IoT, 451
  - clients, 445
  - message structure, 445
  - Spark support, 445–446
  - as transport protocol, 444
- MQTTUtils package, 445–446
- MR1 (MapReduce v1), YARN versus, 19–20
- multi-node Standalone clusters,
  - installing, 36–38
- multiple concurrent applications,
  - scheduling, 469–470
- multiple inheritance in Scala, 155–156
- multiple jobs within applications,
  - scheduling, 470–471
- mutable variables in Scala, 144

## N

- Naive Bayes, 372–373
- NaiveBayes.train method, 372–373
- name value pairs. *See* key value pairs (KVP)
- named functions
  - in Python, 179–180
  - in Scala, 153
- NameNode, 16–17
  - DataNodes and, 17
- naming conventions
  - in Scala, 142
  - for SparkContext, 47

**narrow dependencies**, 109  
**neural networks**, 381  
**newAPIHadoopFile()** method, 128  
**NewHadoopRDDs**, 112  
**Nexus**, 22  
**NodeManagers**, 20–21  
**nodes**. *See also* vertices  
     in clusters, 22–23  
     in DAG, 47  
     DataNodes, 17  
     in decision trees, 368  
     defined, 13  
     EMR types, 74  
     NameNode, 16–17  
**non-deterministic functions, fault tolerance and**, 111  
**non-interactive use of Spark**, 7, 8  
**non-splittable compression formats**, 94, 113, 249  
**NoSQL**  
     Cassandra  
         accessing via Spark, 427–429  
         CQL (Cassandra Query Language), 426–427  
         data model, 426  
         HBase versus, 425–426, 431  
     characteristics of, 418–419, 431  
     DynamoDB, 429–430  
     future of, 430  
     HBase, 419  
         data distribution, 422  
         data model and shell, 420–422  
         reading and writing data with Spark, 423–425

    history of, 417–418  
     implementations of, 430  
     system types, 419, 431  
**notebooks in IPython**, 187–189  
     advantages of, 194  
     kernels and, 189  
     with PySpark, 189–193  
**numeric data type in R**, 345  
**numeric functions**  
     max(), 230  
     mean(), 230  
     min(), 229–230  
     in R, 349  
     stats(), 231–232  
     stdev(), 231  
     sum(), 230–231  
     variance(), 231  
**NumPy library**, 377  
**Nutch**, 11–12, 115

## O

**object comparison in Scala**, 143  
**object files, creating RDDs from**, 99  
**object serialization in Python**, 174  
     JSON, 174–176  
     Pickle, 176–178  
**object stores**, 63  
**objectFile()** method, 99  
**object-oriented programming in Scala**  
     classes and inheritance, 153–155  
     mixin composition, 155–156  
     polymorphism, 157  
     singleton objects, 156–157  
**objects (HDFS), deleting**, 19  
**observations in R**, 352  
**Odersky, Martin**, 137  
**off-heap persistence with Alluxio**, 256  
**OOP**. *See* object-oriented programming in Scala  
**Optimized Row Columnar (ORC)**, 299  
**optimizing**. *See also* performance management  
     applications  
         associative operations, 527–529  
         collecting data, 530  
         diagnosing problems, 536–539  
         dynamic allocation, 531–532  
         with filtering, 527  
         functions and closures, 529–530  
         serialization, 531  
     joins, 221  
     parallelization, 531  
     partitions, 534–535  
**ORC (Optimized Row Columnar)**, 299  
**orc()** method, 300–301, 316  
**orderBy()** method, 313  
**outdegrees**, 400  
**outDegrees method**, 408–409  
**outer joins**, 219  
**output formats in Hadoop**, 251–253  
**output operations for DStreams**, 331–333



## P

### packages

GraphFrames.  
See GraphFrames

in R, 348–349  
datasets package,  
351–352

Spark Packages, 406

### packaging Scala programs, 141

### Page, Larry, 402–403, 414

### PageRank, 402–403, 405

defined, 414  
implementing with  
GraphFrames, 411–413

### pair RDDs, 111, 211

flatMapValues() method,  
213–214  
foldByKey() method, 217  
groupByKey() method,  
215–216, 233  
keyBy() method, 213  
keys() method, 212  
mapValues() method, 213  
reduceByKey() method,  
216–217, 233  
sortByKey() method, 217–218  
subtractByKey() method,  
218–219  
values() method, 212

### parallelization

optimizing, 531  
in Python, 181

### parallelize() method, 105–106

### parent RDDs, 109

### Parquet, 299

writing DataFrame data to,  
315–316

### parquet() method, 299–300, 316

### Partial DAG Execution (PDE), 321

### partition keys

in Cassandra, 426  
in Kinesis Streams, 446

### partitionBy() method, 273–274

### partitioning function in MapReduce, 121

### PartitionPruningRDDs, 112

### partitions

default behavior, 271–272  
foreachPartition() method,  
276–277  
glom() method, 277  
in Kafka, 436  
limitations on creating, 102  
lookup() method, 277  
mapPartitions() method,  
277–278  
optimal number of, 273, 536  
repartitioning, 272–273  
coalesce() method,  
274–275  
partitionBy() method,  
273–274  
repartition() method, 274  
repartitionAndSort-  
WithinPartitions()  
method, 275–276  
sizing, 272, 280, 534–535,  
540

### pattern matching in Scala, 152

### PDE (Partial DAG Execution), 321

### Pérez, Fernando, 183

### performance management.

See also optimizing  
benchmarks, 519–520  
spark-perf, 521–525

Terasort, 520–521

TPC (Transaction  
Processing Performance  
Council), 520  
when to use, 540

canary queries, 525

Datadog, 525–526

diagnosing problems,  
536–539

Project Tungsten, 533

PyPy, 532–533

### perimeter security, 502–503, 517

### persist() method, 108–109, 241, 314

### persistence

of DataFrames, 314  
of DStreams, 331  
of RDDs, 108–109, 240–243  
off-heap persistence, 256

### Pickle, 176–178

### Pickle files, 99

### pickleFile() method, 178

### pipe() method, 278–279

### Pivotal HAWQ, 290

### Pizza, 137

### planning applications, 47

### POJO (Plain Old Java Object)

format, saving H2O models, 396

### policies (security), 503

### polymorphism in Scala, 157

### POSIX (Portable Operating System Interface), 18

### Powered by Spark web page, 3

### pprint() method, 331–332

### precedence of configuration properties, 460–461

### predict function, 357

**predictive analytics, 355–356**

- machine learning.

- See machine learning

- with SparkR. See SparkR

**predictive models**

- building in SparkR, 355–358

- steps in, 361

**Pregel, 402–403****pricing**

- AWS (Amazon Web Services), 64

- Databricks, 81

**primary keys in Cassandra, 426****primitives**

- in Scala, 141

- in Spark SQL, 301–302

**principals**

- in authentication, 503

- in Kerberos, 512, 513

**printSchema method, 410****probability functions in R, 349****producers**

- defined, 434

- in Kafka, 435

- in Kinesis Streams, 448

**profile startup files in IPython, 187****programming interfaces to Spark, 3–5****Project Tungsten, 533****properties, Spark configuration, 457–460, 477**

- managing, 461

- precedence, 460–461

**Psyco, 169****public data sets, 63****pub-sub messaging model, 434–435, 451****.py file extension, 167****Py4J, 170****PyPy, 169, 532–533****PySpark, 4, 170. See also Python**

- dictionaries, 174

- higher-order functions, 194

- JSON object usage, 176

- Jupyter notebooks and, 189–193

- pickleFile() method, 178

- saveAsPickleFile() method, 178

- shell, 6

- tuples, 172

- Word Count algorithm (MapReduce example) in, 132–134

**pysparkling shell, 388–390****Python, 165. See also PySpark**

- architecture, 166–167

- CPython, 167–169

- IronPython, 169

- Jython, 169

- Psyco, 169

- PyPy, 169

- PySpark, 170

- Python.NET, 169

- data structures

- dictionaries, 173–174

- lists, 170, 194

- sets, 170–171

- tuples, 171–173, 194

- functional programming in, 178

- anonymous functions, 179–180

- closures, 181–183

- higher-order functions, 180, 194

- parallelization, 181

- short-circuiting, 181

- tail calls, 180–181

- history of, 166

- installing, 31

- IPython (Interactive Python), 183

- advantages of, 194

- history of, 183–184

- Jupyter notebooks, 187–193

- kernels, 189

- Spark usage with, 184–187

- object serialization, 174

- JSON, 174–176

- Pickle, 176–178

- word count in Spark (listing 1.1), 4

**python directory, 39****Python.NET, 169****Q****queueing messages, 435****quorums in Kafka, 436–437****R****R directory, 39****R programming language, 343–344**

- assignment operator (<-), 344

- data frames, 345, 347–348

- creating SparkR data frames from, 351–352
  - matrices versus, 361
- data structures, 345–347
- data types, 344–345
- datasets package, 351–352
- functions and packages, 348–349
- SparkR. *See* SparkR
- randomSplit function, 369–370**
- range() method, 106**
- RBAC (role-based access control), 503**
- RDDs (Resilient Distributed Datasets), 2, 8**
  - actions, 206
    - collect(), 207
    - count(), 206
    - first(), 208–209
    - foreach(), 210–211, 233
    - take(), 207–208
    - top(), 208
  - aggregate actions, 209
    - fold(), 210
    - reduce(), 209
  - benefits of replication, 257
  - coarse-grained versus fine-grained transformations, 107
  - converting DataFrames to, 301
  - creating DataFrames from, 294–295
  - data sampling, 198–199
    - sample() method, 198–199
    - takeSample() method, 199
  - default partition behavior, 271–272
  - in DStreams, 333
  - EdgeRDD objects, 404–405
  - explained, 91–93, 197–198
  - external storage, 247–248
    - Alluxio, 254–257, 258
    - columnar formats, 253, 299
    - compressed options, 249–250
    - Hadoop input/output formats, 251–253
    - saveAsTextFile() method, 248
    - sequence files, 250
  - fault tolerance, 111
  - functional transformations, 199
    - filter() method, 201–202
    - flatMap() method, 200–201, 232
    - map() method, 199–200, 232, 233
  - GraphRDD objects, 405
  - grouping and sorting data, 202
    - distinct() method, 203–204
    - groupBy() method, 202
    - sortBy() method, 202–203
  - joins, 219
    - cartesian() method, 225–226
    - cogroup() method, 224–225
    - example usage, 226–229
    - fullOuterJoin() method, 223–224
    - join() method, 219–221
    - leftOuterJoin() method, 221–222
    - rightOuterJoin() method, 222–223
    - types of, 219
  - key value pairs (KVP), 211
    - flatMapValues() method, 213–214
    - foldByKey() method, 217
    - groupByKey() method, 215–216, 233
    - keyBy() method, 213
    - keys() method, 212
    - mapValues() method, 213
    - reduceByKey() method, 216–217, 233
    - sortByKey() method, 217–218
    - subtractByKey() method, 218–219
    - values() method, 212
  - lazy evaluation, 107–108
  - lineage, 109–110, 235–237
  - loading data, 93
    - from datasources, 100
    - from JDBC datasources, 100–103
    - from JSON files, 103–105
    - from object files, 99
    - programmatically, 105–106
    - from text files, 93–99
  - numeric functions
    - max(), 230
    - mean(), 230
    - min(), 229–230
    - stats(), 231–232

- stddev(), 231
- sum(), 230–231
- variance(), 231
- off-heap persistence, 256
- persistence, 108–109
- processing with external programs, 278–279
- resilient, explained, 113
- set operations, 204
  - intersection() method, 205
  - subtract() method, 205–206
  - union() method, 204–205
- storage levels, 237
  - caching RDDs, 239–240, 243
  - checkpointing RDDs, 244–247, 258
  - flags, 237–238
  - getStorageLevel() method, 238–239
  - persisting RDDs, 240–243
  - selecting, 239
- Storage tab (application UI), 484–485
- types of, 111–112
- VertexRDD objects, 404
- read command, 348**
- read.csv() method, 348**
- read.fwf() method, 348**
- reading HBase data, 423–425**
- read.jdbc() method, 102–103**
- read.json() method, 104**
- read.table() method, 348**
- realms, 513**
- receivers in Kafka, 437–438, 451**
- recommenders, implementing, 374–375**
- records**
  - defined, 92, 117
  - key value pairs (KVP) and, 118
- Red Hat Linux, installing Spark, 30–31**
- Redis, 430**
- reduce() method, 122, 209**
  - in Python, 170
  - in Word Count algorithm, 129–132
- Reduce phase, 119, 121–122**
- reduceByKey() method, 131, 132, 216–217, 233, 527–529**
- reduceByKeyAndWindow() method, 339**
- reference counting, 169**
- reflection, 302**
- regions (AWS), 62**
- regions in HBase, 422**
- relational databases, creating RDDs from, 100**
- repartition() method, 274, 314**
- repartitionAndSortWithinPartitions() method, 275–276**
- repartitioning, 272–273**
  - coalesce() method, 274–275
  - DataFrames, 314
  - expense of, 215
  - partitionBy() method, 273–274
  - repartition() method, 274
  - repartitionAndSortWithinPartitions() method, 275–276
- replication**
  - benefits of, 257
  - of blocks, 15–16, 25
  - in HDFS, 14–16
- replication factor, 15**
- requirements for Spark installation, 28**
- resilient**
  - defined, 92
  - RDDs as, 113
- Resilient Distributed Datasets (RDDs). See RDDs (Resilient Distributed Datasets)**
- resource management**
  - Dynamic Resource Allocation, 476, 531–532
  - list of alternatives, 22
  - with MapReduce. See MapReduce
  - in Standalone mode, 463
  - with YARN. See YARN (Yet Another Resource Negotiator)
- ResourceManager, 20–21, 471–472**
  - as cluster manager, 51–52
- Riak, 430**
- right outer joins, 219**
- rightOuterJoin() method, 222–223**
- role-based access control (RBAC), 503**
- roles (security), 503**
- RStudio, SparkR usage with, 358–360**
- running applications**
  - in local mode, 56–58
  - on YARN, 20–22, 51, 472–473
    - application management, 473–475
    - ApplicationsMaster, 52–53, 471–472
    - log file management, 56
    - ResourceManager, 51–52

- yarn-client submission mode, 54–55
- yarn-cluster submission mode, 53–54

#### runtime architecture of Python, 166–167

- CPython, 167–169
- IronPython, 169
- Jython, 169
- Psyco, 169
- PyPy, 169
- PySpark, 170
- Python.NET, 169

## S

#### S3 (Simple Storage Service), 63

**sample()** method, 198–199, 309

**sampleBy()** method, 309

#### sampling data, 198–199

- sample()** method, 198–199
- takeSample()** method, 199

#### SASL (Simple Authentication and Security Layer), 506, 509

**save\_model** function, 395

**saveAsHadoopFile()** method, 251–252

**saveAsNewAPIHadoopFile()** method, 253

**saveAsPickleFile()** method, 177–178

**saveAsSequenceFile()** method, 250

**saveAsTable()** method, 315

**saveAsTextFile()** method, 93, 248

**saveAsTextFiles()** method, 332–333

#### saving

- DataFrames to external storage, 314–316
- H2O models, 395–396

#### sbin directory, 39

#### sbt (Simple Build Tool for Scala and Java), 139

#### Scala, 2, 137

- architecture, 139
- comparing objects, 143
- compiling programs, 140–141
- control structures, 149
  - do while and while loops, 151–152
  - for loops, 150–151
  - if expressions, 149–150
  - named functions, 153
  - pattern matching, 152
- data structures, 144
  - lists, 145–146, 163
  - maps, 148–149
  - sets, 146–147, 163
  - tuples, 147–148
- functional programming in
  - anonymous functions, 158
  - closures, 158–159
  - currying, 159
  - first-class functions, 157, 163
  - function literals versus function values, 163
  - higher-order functions, 158
  - immutable data structures, 160
  - lazy evaluation, 160
- history of, 137–138
- installing, 31, 139–140

naming conventions, 142

object-oriented programming in

- classes and inheritance, 153–155
- mixin composition, 155–156
- polymorphism, 157
- singleton objects, 156–157

packaging programs, 141

primitives, 141

shell, 6

type inference, 144

value classes, 142–143

variables, 144

Word Count algorithm example, 160–162

word count in Spark (listing 1.2), 4

#### scalability of Spark, 2

#### scalac compiler, 139

#### scheduling

- application tasks, 47
- in Standalone mode, 469
  - multiple concurrent applications, 469–470
  - multiple jobs within applications, 470–471
- with YARN. *See* YARN (Yet Another Resource Negotiator)

#### schema-on-read systems, 12

#### SchemaRDDs. *See* DataFrames

#### schemas for DataFrames

- defining, 304
- inferring, 302–304

#### schemes in URIs, 95

**Secure Sockets Layer (SSL), 506–510****security, 501–502**

- authentication, 503–504
  - encryption, 506–510
  - shared secrets, 504–506
- authorization, 503–504
- gateway services, 503
- Java Servlet Filters, 510–512, 517
- Kerberos, 512–514, 517
  - client commands, 514
  - configuring, 515–516
  - with Hadoop, 514–515
  - terminology, 513
- perimeter security, 502–503, 517

**security groups, 62****select() method, 309, 322****selecting**

- Spark deployment modes, 43
- storage levels for RDDs, 239

**sequence files**

- creating RDDs from, 99
- external storage, 250

**sequenceFile() method, 99****SequenceFileRDDs, 111****serialization**

- optimizing applications, 531
- in Python, 174
  - JSON, 174–176
  - Pickle, 176–178

**service ticket, 513****set operations, 204**

- for DataFrames, 311–314
- intersection() method, 205
- subtract() method, 205–206
- union() method, 204–205

**setCheckpointDir() method, 244****sets**

- in Python, 170–171
- in Scala, 146–147, 163

**severity levels in Log4j framework, 493****shards in Kinesis Streams, 446****shared nothing, 15, 92****shared secrets, 504–506****shared variables.**

- See accumulators; broadcast variables

**Shark, 283–284****shells**

- Cassandra, 426–427
- HBase, 420–422
- interactive Spark usage, 5–7, 8
- pysparkling, 388–390
- SparkR, 350–351

**short-circuiting in Python, 181****show() method, 306****shuffle, 108**

- diagnosing performance problems, 536–538
- expense of, 215

**Shuffle phase, 119, 121, 135****ShuffledRDDs, 112****side effects of functions, 181****Simple Authentication and Security Layer (SASL), 506, 509****Simple Storage Service (S3), 63****SIMR (Spark In MapReduce), 22****single master mode (Alluxio), 254–255****single point of failure (SPOF), 38****singleton objects in Scala, 156–157****sizing partitions, 272, 280, 534–535, 540****slave nodes**

- defined, 23
- starting in Standalone mode, 463
- worker UIs, 463–466

**sliding window operations with DStreams, 337–339, 340****slots (MapReduce), 20****Snappy, 94****socketTextStream() method, 327–328****Solr, 430****sortBy() method, 202–203****sortByKey() method, 217–218****sorting data, 202**

- distinct() method, 203–204
- foldByKey() method, 217
- groupBy() method, 202
- groupByKey() method, 215–216, 233
- orderBy() method, 313
- reduceByKey() method, 216–217, 233
- sortBy() method, 202–203
- sortByKey() method, 217–218
- subtractByKey() method, 218–219

**sources. See data sources****Spark**

- as abstraction, 2
- application support, 3
- application UI. See application UI
- Cassandra access, 427–429
- configuring
  - broadcast variables, 262
  - configuration properties, 457–460, 477

- environment variables, 454–457
- managing configuration, 461
- precedence, 460–461
- defined, 1–2
- deploying
  - on Databricks, 81–88
  - on EC2, 64–73
  - on EMR, 73–80
- deployment modes. *See also* Spark on YARN deployment mode; Spark Standalone deployment mode
  - list of, 27–28
  - selecting, 43
- downloading, 29–30
- Hadoop and, 2, 8
  - HDFS as data source, 24
  - YARN as resource scheduler, 24
- input/output types, 7
- installing
  - on Hadoop, 39–42
  - on Mac OS X, 33–34
  - on Microsoft Windows, 34–36
  - as multi-node Standalone cluster, 36–38
  - on Red Hat/Centos, 30–31
  - requirements for, 28
  - in Standalone mode, 29–36
  - subdirectories of installation, 38–39
  - on Ubuntu/Debian Linux, 32–33
- interactive use, 5–7, 8

- IPython usage, 184–187
- Kafka support, 437
  - direct stream access, 438, 451
  - KafkaUtils package, 439–443
  - receivers, 437–438, 451
- Kinesis Streams support, 448–450
- logging. *See* logging
- machine learning in, 367
- MapReduce versus, 2, 8
- master UI, 487
- metrics, collecting, 490–492
- MQTT support, 445–446
- non-interactive use, 7, 8
- programming interfaces to, 3–5
- scalability of, 2
- security. *See* security

### **Spark applications. *See* applications**

### **Spark History Server, 488**

- API access, 489–490
- configuring, 488
- deploying, 488
- diagnosing performance problems, 539
- UI (user interface) for, 488–489

### **Spark In MapReduce (SIMR), 22**

### **Spark ML, 367**

- Spark MLlib versus, 378

### **Spark MLlib, 367**

- classification in, 367
- decision trees, 368–372
- Naive Bayes, 372–373

- clustering in, 375–377
- collaborative filtering in, 373–375
- Spark ML versus, 378

### **Spark on YARN deployment mode, 27–28, 39–42, 471–473**

- application management, 473–475
- environment variables, 456–457
- scheduling, 475–476

### **Spark Packages, 406**

### **Spark SQL, 283**

- accessing
  - via Beeline, 318–321
  - via external applications, 319
  - via JDBC/ODBC interface, 317–318
  - via spark-sql shell, 316–317
- architecture, 290–292
- DataFrames, 294
  - built-in functions, 310
  - converting to RDDs, 301
  - creating from Hive tables, 295–296
  - creating from JSON objects, 296–298
  - creating from RDDs, 294–295
  - creating with DataFrameReader, 298–301
- data model, 301–302
- defining schemas, 304
- functional operations, 306–310

- inferring schemas, 302–304
  - metadata operations, 305–306
  - saving to external storage, 314–316
  - set operations, 311–314
  - UDFs (user-defined functions), 310–311
- history of, 283–284
- Hive and, 291–292
- HiveContext, 292–293, 322
- SQLContext, 292–293, 322
- Spark SQL DataFrames**
  - caching, persisting, repartitioning, 314
- Spark Standalone deployment mode, 27–28, 29–36, 461–462**
  - application management, 466–469
  - daemon environment variables, 455–456
  - on Mac OS X, 33–34
  - master and worker UIs, 463–466
  - on Microsoft Windows, 34–36
  - as multi-node Standalone cluster, 36–38
  - on Red Hat/Centos, 30–31
  - resource allocation, 463
  - scheduling, 469
    - multiple concurrent applications, 469–470
    - multiple jobs within applications, 470–471
  - starting masters/slaves, 463
  - on Ubuntu/Debian Linux, 32–33
- Spark Streaming**
  - architecture, 324–325
  - DStreams, 326–327
    - broadcast variables and accumulators, 331
  - caching and persistence, 331
  - checkpointing, 330–331, 340
  - data sources, 327–328
  - lineage, 330
  - output operations, 331–333
  - sliding window operations, 337–339, 340
  - state operations, 335–336, 340
  - transformations, 328–329
  - history of, 323–324
  - StreamingContext, 325–326
  - word count example, 334–335
- SPARK\_HOME variable, 454**
- SparkContext, 46–47**
- spark-ec2 shell script, 65**
  - actions, 65
  - options, 66
  - syntax, 65
- spark-env.sh script, 454**
- Sparkling Water, 387, 397**
  - architecture, 387–388
  - example exercise, 393–395
  - H2OFrames, 390–393
  - pysparkling shell, 388–390
- spark-perf, 521–525**
- SparkR**
  - building predictive models, 355–358
- creating data frames
  - from CSV files, 352–354
  - from Hive tables, 354–355
  - from R data frames, 351–352
- documentation, 350
- RStudio usage with, 358–360
- shell, 350–351
- spark-sql shell, 316–317**
- spark-submit command, 7, 8**
  - master local argument, 59
- sparsity, 421**
- speculative execution, 135, 280**
  - defined, 21
  - in MapReduce, 124
- splittable compression formats, 94, 113, 249**
- SPOF (single point of failure), 38**
- spot instances, 62**
- SQL (Structured Query Language), 283. See also Hive; Spark SQL**
- sql() method, 295–296
- SQL on Hadoop, 289–290
- SQLContext, 100, 292–293, 322
- SSL (Secure Sockets Layer), 506–510**
- stages**
  - in DAG, 47
  - diagnosing performance problems, 536–538
  - tasks and, 59
- Stages tab (application UI), 483–484, 499**
- Standalone mode. See Spark Standalone deployment mode**
- starting masters/slaves in Standalone mode, 463**



state operations with DStreams,  
335–336, 340

statistical functions

- max(), 230
- mean(), 230
- min(), 229–230
- in R, 349
- stats(), 231–232
- stdev(), 231
- sum(), 230–231
- variance(), 231

stats() method, 231–232

stdev() method, 231

stemming, 128

step execution mode (EMR), 74

stopwords, 128

storage levels for RDDs, 237

- caching RDDs, 239–240, 243
- checkpointing RDDs,  
244–247, 258
- external storage, 247–248
  - Alluxio, 254–257, 258
  - columnar formats, 253,  
299
  - compressed options,  
249–250
  - Hadoop input/output  
formats, 251–253
  - saveAsTextFile() method,  
248
  - sequence files, 250
- flags, 237–238
- getStorageLevel() method,  
238–239
- persisting RDDs, 240–243
- selecting, 239

Storage tab (application UI),  
484–485, 499

StorageClass constructor, 238

Storm, 323

stream processing. *See also*

messaging systems

- DStreams, 326–327
  - broadcast variables and  
accumulators, 331
  - caching and persistence,  
331
  - checkpointing, 330–331,  
340
  - data sources, 327–328
  - lineage, 330
  - output operations,  
331–333
  - sliding window operations,  
337–339, 340
  - state operations,  
335–336, 340
  - transformations, 328–329
- Spark Streaming
  - architecture, 324–325
  - history of, 323–324
  - StreamingContext,  
325–326
  - word count example,  
334–335

StreamingContext, 325–326

StreamingContext.checkpoint()  
method, 330

streams in Kinesis, 446–447

strict evaluation, 160

Structured Query Language (SQL),  
283. *See also* Hive; Spark SQL

subdirectories of Spark  
installation, 38–39

subgraphs, 410

subtract() method, 205–206, 313

subtractByKey() method, 218–219

sum() method, 230–231

summary function, 357, 392

supervised learning, 355

## T

table() method, 296

tables

- in Cassandra, 426
- in Databricks, 81
- in Hive
  - creating DataFrames from,  
295–296
  - creating SparkR data  
frames from, 354–355
  - internal versus external,  
289
  - writing DataFrame data  
to, 315

tablets (Bigtable), 422

Tachyon. *See* Alluxio

tail call recursion in Python,  
180–181

tail calls in Python, 180–181

take() method, 207–208, 306, 530

takeSample() method, 199

task attempts, 21

task nodes, core nodes versus, 89

tasks

- in DAG, 47
- defined, 20–21
- diagnosing performance  
problems, 536–538
- scheduling, 47
- stages and, 59

**Terasort, 520–521**

**Term Frequency-Inverse Document Frequency (TF-IDF), 367**

**test data sets, 369–370**

**text files**

- creating DataFrames from, 298–299
- creating RDDs from, 93–99
- saving DStreams as, 332–333

**text input format, 127**

**text() method, 298–299**

**textFile() method, 95–96**

- text input format, 128
- wholeTextFiles() method versus, 97–99

**textFileStream() method, 328**

**Tez, 289**

**TF-IDF (Term Frequency-Inverse Document Frequency), 367**

**Thrift JDBC/ODBC server, accessing Spark SQL, 317–318**

**ticket granting service (TGS), 513**

**ticket granting ticket (TGT), 513**

**tokenization, 127**

**top() method, 208**

**topic filtering, 434–435, 451**

**TPC (Transaction Processing Performance Council), 520**

**training data sets, 369–370**

**traits in Scala, 155–156**

**Transaction Processing Performance Council (TPC), 520**

**transformations**

- cartesian(), 225–226
- coarse-grained versus fine-grained, 107
- cogroup(), 224–225

defined, 47

distinct(), 203–204

for DStreams, 328–329

filter(), 201–202

flatMap(), 131, 200–201  
map() versus, 135, 232

flatMapValues(), 213–214

foldByKey(), 217

fullOuterJoin(), 223–224

groupBy(), 202

groupByKey(), 215–216, 233

intersection(), 205

join(), 219–221

keyBy(), 213

keys(), 212

lazy evaluation, 107–108

leftOuterJoin(), 221–222

lineage, 109–110, 235–237

map(), 130, 199–200  
flatMap() versus, 135, 232  
foreach() action versus, 233

passing functions to, 540–541

mapValues(), 213

of RDDs, 92

reduceByKey(), 131, 132, 216–217, 233

rightOuterJoin(), 222–223

sample(), 198–199

sortBy(), 202–203

sortByKey(), 217–218

subtract(), 205–206

subtractByKey(), 218–219

union(), 204–205

values(), 212

**transport protocol, MQTT as, 444**

**Trash settings in HDFS, 19**

**triangle count algorithm, 405**

**triplets, 402**

**tuple extraction in Scala, 152**

**tuples, 132**

in Python, 171–173, 194

in Scala, 147–148

**type inference in Scala, 144**

**Typesafe, Inc., 138**

## U

**Ubuntu Linux, installing Spark, 32–33**

**udf() method, 311**

**UDFs (user-defined functions) for DataFrames, 310–311**

**UI (user interface).**

See application UI

**Uniform Resource Identifiers (URIs), schemes in, 95**

**union() method, 204–205**

**unionAll() method, 313**

**UnionRDDs, 112**

**unnamed functions**

- in Python, 179–180
- in Scala, 158

**unpersist() method, 241, 262, 314**

**unsupervised learning, 355**

**updateStateByKey() method, 335–336**

**uploading (ingesting) files, 18**

**URIs (Uniform Resource Identifiers), schemes in, 95**

user interface (UI).

See application UI

user-defined functions (UDFs) for DataFrames, 310–311

## V

value classes in Scala, 142–143

value() method

accumulators, 266

broadcast variables, 261–262

values, 118

values() method, 212

van Rossum, Guido, 166

variables

accumulators, 265–266

accumulator() method, 266

custom accumulators, 267

usage example, 268–270

value() method, 266

warning about, 268

bound variables, 158

broadcast variables, 259–260

advantages of, 263–265, 280

broadcast() method, 260–261

configuration options, 262

unpersist() method, 262

usage example, 268–270

value() method, 261–262

environment variables, 454

cluster application

deployment, 457

cluster manager

independent variables, 454–455

Hadoop-related, 455

Spark on YARN

environment variables, 456–457

Spark Standalone daemon, 455–456

free variables, 158

in R, 352

in Scala, 144

variance() method, 231

vectors in R, 345–347

VertexRDD objects, 404

vertices

creating vertex DataFrames, 407

in DAG, 47

defined, 399

indegrees, 400

outdegrees, 400

vertices method, 407–408

VPC (Virtual Private Cloud), 62

## W

WAL (write ahead log), 435

weather dataset, 368

web interface for H2O,

382–383

websites, Powered by Spark, 3

WEKA machine learning software package, 368

while loops in Scala, 151–152

wholeTextFiles() method, 97

textFile() method versus, 97–99

wide dependencies, 110

window() method, 337–338

windowed DStreams, 337–339, 340

Windows, installing Spark, 34–36

Word Count algorithm

(MapReduce example), 126

map() and reduce() methods, 129–132

operational overview, 127–129

in PySpark, 132–134

reasons for usage, 126–127

in Scala, 160–162

word count in Spark

using Java (listing 1.3), 4–5

using Python (listing 1.1), 4

using Scala (listing 1.2), 4

workers, 45, 48–49

executors versus, 59

worker UIs, 463–466

WORM (Write Once Read Many), 14

write ahead log (WAL), 435

writing HBase data, 423–425

## Y

Yahoo! in history of big data, 11–12

YARN (Yet Another Resource Negotiator), 12

executor logs, 497

explained, 19–20

reasons for development, 25

running applications, 20–22, 51

ApplicationsMaster, 52–53

log file management, 56

- ResourceManager, 51–52
  - yarn-client submission mode, 54–55
  - yarn-cluster submission mode, 53–54
- running H2O with, 384–386
- Spark on YARN deployment mode, 27–28, 39–42, 471–473
  - application management, 473–475
  - environment variables, 456–457

- scheduling, 475–476
  - as Spark resource scheduler, 24
- YARN Timeline Server UI, 56**
- yarn-client submission mode, 42, 43, 54–55
- yarn-cluster submission mode, 41–42, 43, 53–54
- Yet Another Resource Negotiator (YARN). See YARN (Yet Another Resource Negotiator)**
- yield operator in Scala, 151

## Z

- Zeppelin, 75**
- Zharia, Matei, 1**
- Zookeeper, 38, 436**
  - installing, 441