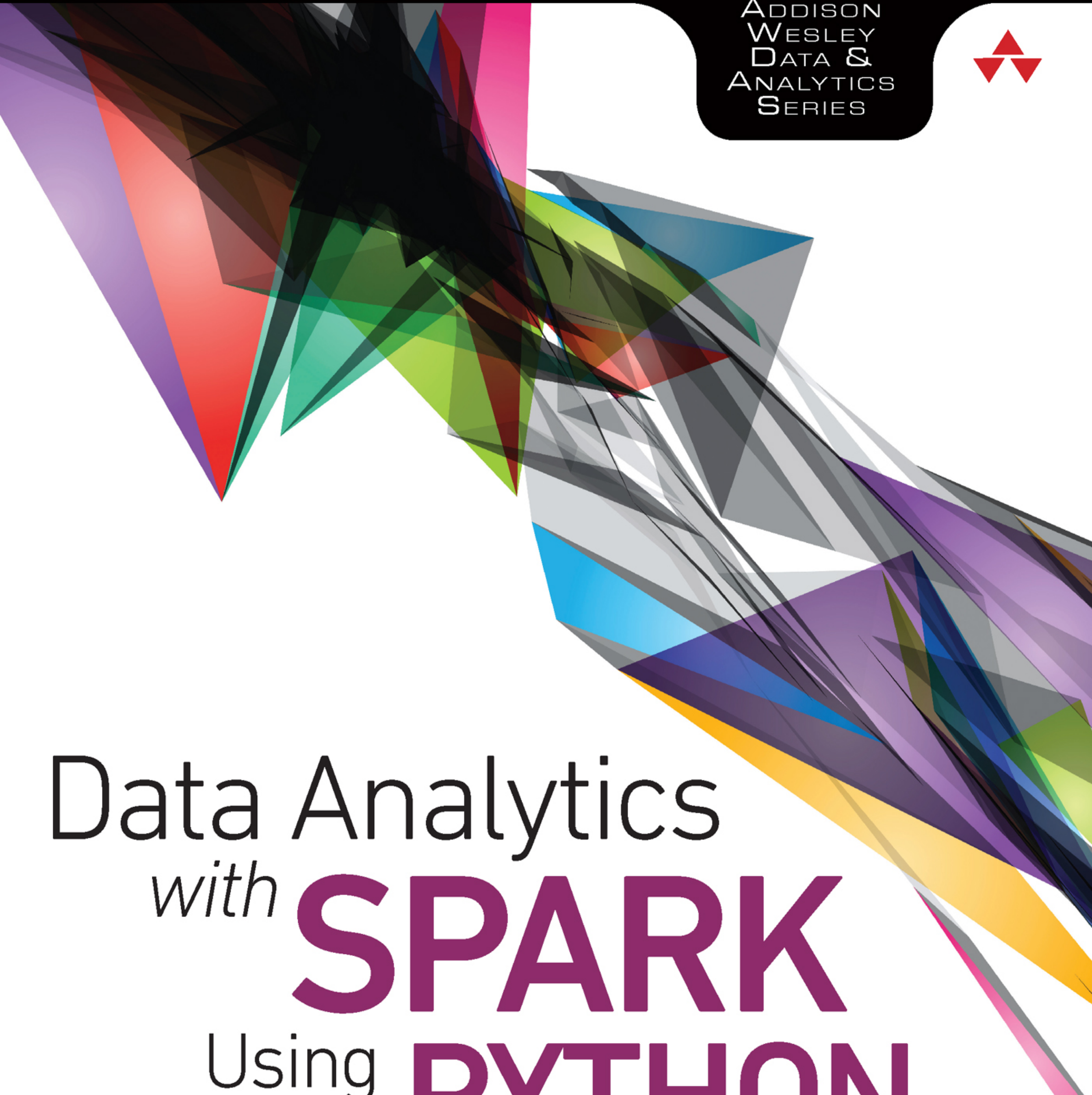
Addison Wesley Data & Analytics Series

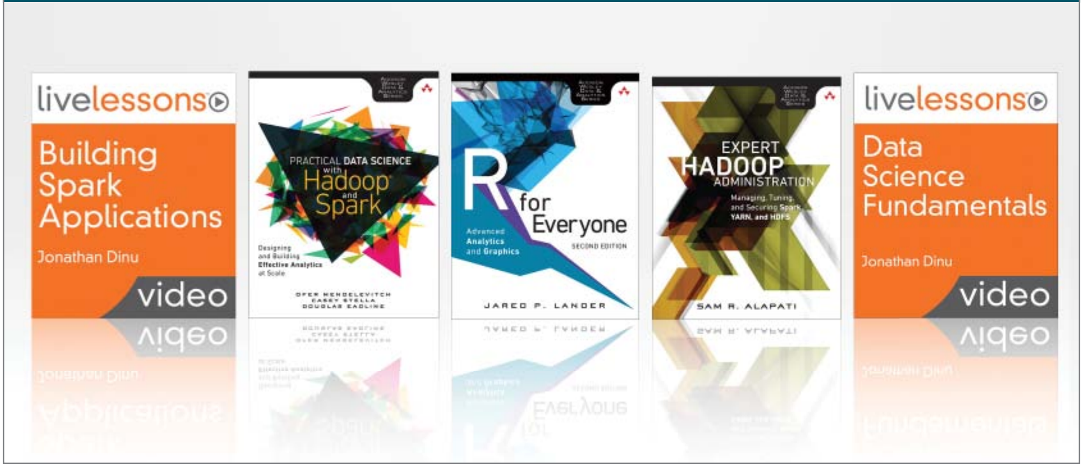


Data Analytics with SPARK Using PYTHON

Jeffrey Aven

Data Analytics with Spark Using Python

The Pearson Addison-Wesley Data and Analytics Series



Visit informit.com/awdataseries for a complete list of available publications.

The Pearson Addison-Wesley Data and Analytics Series provides readers with practical knowledge for solving problems and answering questions with data. Titles in this series primarily focus on three areas:

1. **Infrastructure:** how to store, move, and manage data
2. **Algorithms:** how to mine intelligence or make predictions based on data
3. **Visualizations:** how to represent data and insights in a meaningful and compelling way

The series aims to tie all three of these areas together to help the reader build end-to-end systems for fighting spam; making recommendations; building personalization; detecting trends, patterns, or problems; and gaining insight from the data exhaust of systems and user interactions.

Make sure to connect with us!
informit.com/socialconnect

Pearson Addison-Wesley

informIT.com
the trusted technology learning source

O'REILLY® Safari

Data Analytics with Spark Using Python

Jeffrey Aven

Addison-Wesley

Boston • Columbus • Indianapolis • New York • San Francisco • Amsterdam
Cape Town • Dubai • London • Madrid • Milan • Munich • Paris
Montreal • Toronto • Delhi • Mexico City • São Paulo • Sydney
Hong Kong • Seoul • Singapore • Taipei • Tokyo

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

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 or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2018938456

© 2018 Pearson Education, Inc.

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms, and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit www.pearsoned.com/permissions/.

Microsoft and/or its respective suppliers make no representations about the suitability of the information contained in the documents and related graphics published as part of the services for any purpose. All such documents and related graphics are provided “as is” without warranty of any kind. Microsoft and/ or its respective suppliers hereby disclaim all warranties and conditions with regard to this information, including all warranties and conditions of merchantability, whether express, implied or statutory, fitness for a particular purpose, title and non-infringement. In no event shall Microsoft and/or its respective suppliers be liable for any special, indirect or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortious action, arising out of or in connection with the use or performance of information available from the services. The documents and related graphics contained herein could include technical inaccuracies or typographical errors. Changes are periodically added to the information herein. Microsoft and/or its respective suppliers may make improvements and/or changes in the product(s) and/or the program(s) described herein at any time. Partial screenshots may be viewed in full within the software version specified.

Microsoft® Windows®, and Microsoft Office® are registered trademarks of the Microsoft Corporation in the U.S.A. and other countries. This book is not sponsored or endorsed by or affiliated with the Microsoft Corporation.

ISBN-13: 978-0-13-484601-9
ISBN-10: 0-13-484601-X

1 18

Editor-in-Chief
Greg Wiegand

Executive Editor
Trina MacDonald

Development Editor
Amanda Kaufmann

Managing Editor
Sandra Schroeder

Senior Project Editor
Lori Lyons

Technical Editor
Yaniv Rodenski

Copy Editor
Catherine D. Wilson

Project Manager
Dhayanidhi Karunanidhi

Indexer
Erika Millen

Proofreader
Jeanine Furino

Cover Designer
Chuti Prasertsith

Compositor
codemantra

Contents at a Glance

Preface **xi**

Introduction **1**

I: Spark Foundations

1 Introducing Big Data, Hadoop, and Spark **5**

2 Deploying Spark **27**

3 Understanding the Spark Cluster Architecture **45**

4 Learning Spark Programming Basics **59**

II: Beyond the Basics

5 Advanced Programming Using the Spark Core API **111**

6 SQL and NoSQL Programming with Spark **161**

7 Stream Processing and Messaging Using Spark **209**

8 Introduction to Data Science and Machine Learning Using Spark **243**

Index **281**

Table of Contents

Preface xi

Introduction 1

I: Spark Foundations

1 Introducing Big Data, Hadoop, and Spark 5

Introduction to Big Data, Distributed Computing, and Hadoop 5
- A Brief History of Big Data and Hadoop 6
- Hadoop Explained 7

Introduction to Apache Spark 13
- Apache Spark Background 13
- Uses for Spark 14
- Programming Interfaces to Spark 14
- Submission Types for Spark Programs 14
- Input/Output Types for Spark Applications 16
- The Spark RDD 16
- Spark and Hadoop 16

Functional Programming Using Python 17
- Data Structures Used in Functional Python Programming 17
- Python Object Serialization 20
- Python Functional Programming Basics 23

Summary 25

2 Deploying Spark 27

Spark Deployment Modes 27
- Local Mode 28
- Spark Standalone 28
- Spark on YARN 29
- Spark on Mesos 30

Preparing to Install Spark 30

Getting Spark 31

Installing Spark on Linux or Mac OS X 32

Installing Spark on Windows 34

Exploring the Spark Installation 36

Deploying a Multi-Node Spark Standalone Cluster 37

Deploying Spark in the Cloud 39
Amazon Web Services (AWS) 39
Google Cloud Platform (GCP) 41
Databricks 42
Summary 43

3 Understanding the Spark Cluster Architecture 45
Anatomy of a Spark Application 45
Spark Driver 46
Spark Workers and Executors 49
The Spark Master and Cluster Manager 51
Spark Applications Using the Standalone Scheduler 53
Spark Applications Running on YARN 53
Deployment Modes for Spark Applications Running on YARN 53
Client Mode 54
Cluster Mode 55
Local Mode Revisited 56
Summary 57

4 Learning Spark Programming Basics 59
Introduction to RDDs 59
Loading Data into RDDs 61
Creating an RDD from a File or Files 61
Methods for Creating RDDs from a Text File or Files 63
Creating an RDD from an Object File 66
Creating an RDD from a Data Source 66
Creating RDDs from JSON Files 69
Creating an RDD Programmatically 71
Operations on RDDs 72
Key RDD Concepts 72
Basic RDD Transformations 77
Basic RDD Actions 81
Transformations on PairRDDs 85
MapReduce and Word Count Exercise 92
Join Transformations 95
Joining Datasets in Spark 100
Transformations on Sets 103
Transformations on Numeric RDDs 105
Summary 108

II: Beyond the Basics

5 Advanced Programming Using the Spark Core API 111

Shared Variables in Spark 111

Broadcast Variables 112

Accumulators 116

Exercise: Using Broadcast Variables and Accumulators 119

Partitioning Data in Spark 120

Partitioning Overview 120

Controlling Partitions 121

Repartitioning Functions 123

Partition-Specific or Partition-Aware API Methods 125

RDD Storage Options 127

RDD Lineage Revisited 127

RDD Storage Options 128

RDD Caching 131

Persisting RDDs 131

Choosing When to Persist or Cache RDDs 134

Checkpointing RDDs 134

Exercise: Checkpointing RDDs 136

Processing RDDs with External Programs 138

Data Sampling with Spark 139

Understanding Spark Application and Cluster Configuration 141

Spark Environment Variables 141

Spark Configuration Properties 145

Optimizing Spark 148

Filter Early, Filter Often 149

Optimizing Associative Operations 149

Understanding the Impact of Functions and Closures 151

Considerations for Collecting Data 152

Configuration Parameters for Tuning and Optimizing Applications 152

Avoiding Inefficient Partitioning 153

Diagnosing Application Performance Issues 155

Summary 159

6 SQL and NoSQL Programming with Spark 161

Introduction to Spark SQL 161

Introduction to Hive 162

Spark SQL Architecture 166

Getting Started with DataFrames 168
Using DataFrames 179
Caching, Persisting, and Repartitioning DataFrames 187
Saving DataFrame Output 188
Accessing Spark SQL 191
Exercise: Using Spark SQL 194
Using Spark with NoSQL Systems 195
Introduction to NoSQL 196
Using Spark with HBase 197
Exercise: Using Spark with HBase 200
Using Spark with Cassandra 202
Using Spark with DynamoDB 204
Other NoSQL Platforms 206
Summary 206

7 Stream Processing and Messaging Using Spark 209
Introducing Spark Streaming 209
Spark Streaming Architecture 210
Introduction to DStreams 211
Exercise: Getting Started with Spark Streaming 218
State Operations 219
Sliding Window Operations 221
Structured Streaming 223
Structured Streaming Data Sources 224
Structured Streaming Data Sinks 225
Output Modes 226
Structured Streaming Operations 227
Using Spark with Messaging Platforms 228
Apache Kafka 229
Exercise: Using Spark with Kafka 234
Amazon Kinesis 237
Summary 240

8 Introduction to Data Science and Machine Learning Using Spark 243
Spark and R 243
Introduction to R 244
Using Spark with R 250
Exercise: Using RStudio with SparkR 257

Machine Learning with Spark 259
    Machine Learning Primer 259
    Machine Learning Using Spark MLlib 262
    Exercise: Implementing a Recommender Using Spark MLlib 267
    Machine Learning Using Spark ML 271
Using Notebooks with Spark 275
    Using Jupyter (IPython) Notebooks with Spark 275
    Using Apache Zeppelin Notebooks with Spark 278
Summary 279

Index 281

Preface

Spark is at the heart of the disruptive Big Data and open source software revolution. The interest in and use of Spark have grown exponentially, with no signs of abating. This book will prepare you, step by step, for a prosperous career in the Big Data analytics field.

Focus of the Book

This book focuses on the fundamentals of the Spark project, starting from the core and working outward into Spark's various extensions, related or subprojects, and the broader ecosystem of open source technologies such as Hadoop, Kafka, Cassandra, and more.

Although the foundational understanding of Spark concepts covered in this book—including the runtime, cluster and application architecture—are language independent and agnostic, the majority of the programming examples and exercises in this book are written in Python. The Python API for Spark (PySpark) provides an intuitive programming environment for data analysts, data engineers, and data scientists alike, offering developers the flexibility and extensibility of Python with the distributed processing power and scalability of Spark.

The scope of this book is quite broad, covering aspects of Spark from core Spark programming to Spark SQL, Spark Streaming, machine learning, and more. This book provides a good introduction and overview for each topic—enough of a platform for you to build upon any particular area or discipline within the Spark project.

Who Should Read This Book

This book is intended for data analysts and engineers looking to enter the Big Data space or consolidate their knowledge in this area. The demand for engineers with skills in Big Data and its preeminent processing framework, Spark, is exceptionally high at present. This book aims to prepare readers for this growing employment market and arm them with the skills employers are looking for.

Python experience is useful but not strictly necessary for readers of this book as Python is quite intuitive for anyone with any programming experience whatsoever. A good working knowledge of data analysis and manipulation would also be helpful. This book is especially well suited to data warehouse professionals interested in expanding their careers into the Big Data area.

How to Use This Book

This book is structured into two parts and eight chapters. Part I, "Spark Foundations," includes four chapters designed to build a solid understanding of what Spark is, how to deploy Spark, and how to use Spark for basic data processing operations:

- Chapter 1, "Introducing Big Data, Hadoop and Spark," provides a good overview of the Big Data ecosystem, including the genesis and evolution of the Spark project. Key properties of the Spark project are discussed, including what Spark is and how it is used, as well as how Spark relates to the Hadoop project.
- Chapter 2, "Deploying Spark," demonstrates how to deploy a Spark cluster, including the various Spark cluster deployment modes and the different ways you can leverage Spark.

- Chapter 3, "Understanding the Spark Cluster Architecture," discusses how Spark clusters and applications operate, providing a solid understanding of exactly how Spark works.
- Chapter 4, "Learning Spark Programming Basics," focuses on the basic programming building blocks of Spark using the Resilient Distributed Dataset (RDD) API.

Part II, "Beyond the Basics," includes the final four chapters, which extend beyond the Spark core into its uses with SQL and NoSQL systems, streaming applications, and data science and machine learning:

- Chapter 5, "Advanced Programming Using the Spark Core API," covers advanced constructs used to extend, accelerate, and optimize Spark routines, including different shared variables and RDD storage and partitioning concepts and implementations.
- Chapter 6, "SQL and NoSQL Programming with Spark," discusses Spark's integration into the vast SQL landscape as well as its integration with non-relational stores.
- Chapter 7, "Stream Processing and Messaging Using Spark," introduces the Spark streaming project and the fundamental DStream object. It also covers Spark's use with popular messaging systems such as Apache Kafka.
- Chapter 8, "Introduction to Data Science and Machine Learning Using Spark," provides an introduction to predictive modeling using Spark with R as well as the Spark MLlib subproject used to implement machine learning with Spark.

Book Conventions

Key terms or concepts are highlighted in italic. Code, object, and file references are displayed in a monospaced font.

Step-by-step exercises are provided to consolidate each topic.

Accompanying Code and Data for the Exercises

Sample data and source code for each of the exercises in this book is available at **http://sparkusingpython.com**. You can also view or clone the GitHub repository for this book at https://github.com/sparktraining/spark_using_python.

Register This Book

Register your copy of *Data Analytics with Spark Using Python* on the InformIT site for convenient access to updates and/or corrections as they become available. To start the registration process, go to informit.com/register and log in or create an account. Enter the product ISBN (9780134846019) and click Submit. Look on the Registered Products tab for an Access Bonus Content link next to this product, and follow that link to access any available bonus materials. If you would like to be notified of exclusive offers on new editions and updates, please check the box to receive email from us.

About the Author

Jeffrey Aven is an independent Big Data, open source software and cloud computing professional based out of Melbourne, Australia. Jeffrey is a highly regarded consultant and instructor and has authored several other books including *Teach Yourself Apache Spark in 24 Hours* and *Teach Yourself Hadoop in 24 Hours*.

This page intentionally left blank

3

Understanding the Spark Cluster Architecture

It is not the beauty of a building you should look at; it's the construction of the foundation that will stand the test of time.

David Allan Coe, American songwriter

In This Chapter:

- Detailed overview of the Spark application and cluster components
- Spark resource schedulers and Cluster Managers
- How Spark applications are scheduled on YARN clusters
- Spark deployment modes

Before you begin your journey as a Spark programmer, you should have a solid understanding of the Spark application architecture and how applications are executed on a Spark cluster. This chapter closely examines the components of a Spark application, looks at how these components work together, and looks at how Spark applications run on Standalone and YARN clusters.

Anatomy of a Spark Application

A Spark application contains several components, all of which exist whether you're running Spark on a single machine or across a cluster of hundreds or thousands of nodes.

Each component has a specific role in executing a Spark program. Some of these roles, such as the client components, are passive during execution; other roles are active in the execution of the program, including components executing computation functions.

The components of a Spark application are the *Driver*, the *Master*, the *Cluster Manager*, and the *Executor(s)*, which run on worker nodes, or *Workers*. Figure 3.1 shows all the Spark components in the context of a Spark Standalone application. You will learn more about each component and its function in more detail later in this chapter.

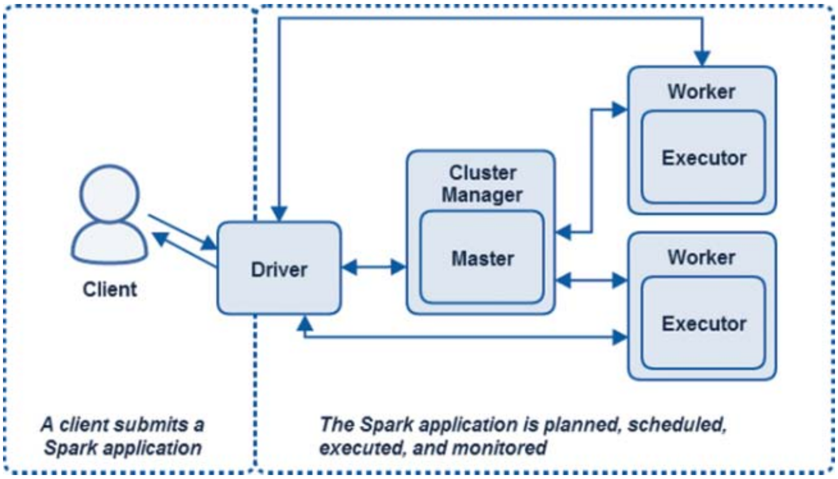

Worker
Executor
Cluster Manager
Master
Client
Driver
Worker
Executor
A client submits a Spark application
The Spark application is planned, scheduled, executed, and monitored

Figure 3.1 Spark Standalone cluster application components.

All Spark components, including the Driver, Master, and Executor processes, run in Java virtual machines (JVMs). A JVM is a cross-platform runtime engine that can execute instructions compiled into Java bytecode. Scala, which Spark is written in, compiles into bytecode and runs on JVMs.

It is important to distinguish between Spark's runtime application components and the locations and node types on which they run. These components run in different places using different deployment modes, so don't think of these components in physical node or instance terms. For instance, when running Spark on YARN, there would be several variations of Figure 3.1. However, all the components pictured are still involved in the application and have the same roles.

Spark Driver

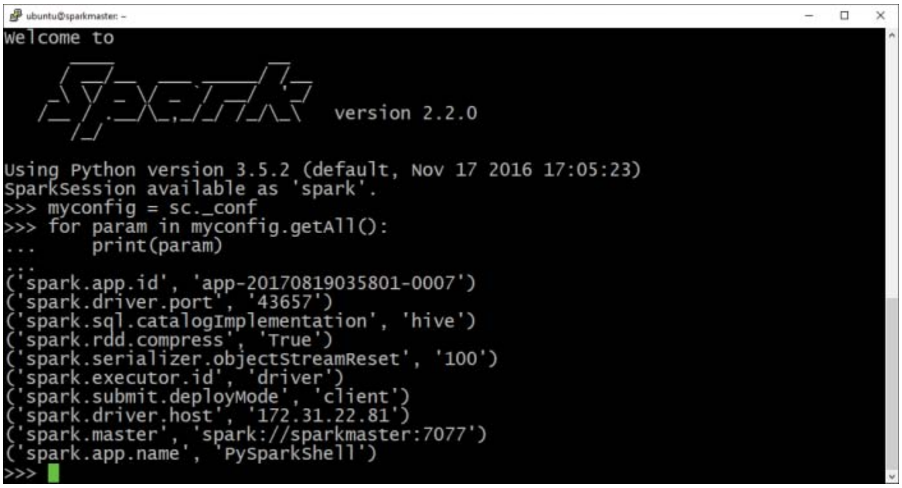
The life of a Spark application starts and finishes with the Spark Driver. The Driver is the process that clients use to submit applications in Spark. The Driver is also responsible for planning and coordinating the execution of the Spark program and returning status and/or results (data) to the client. The Driver can physically reside on a client or on a node in the cluster, as you will see later.

SparkSession

The Spark Driver is responsible for creating the *SparkSession*. The SparkSession object represents a connection to a Spark cluster. The SparkSession is instantiated at the beginning of a Spark application, including the interactive shells, and is used for the entirety of the program.

Prior to Spark 2.0, entry points for Spark applications included the SparkContext, used for Spark core applications; the SQLContext and HiveContext, used with Spark SQL applications; and the StreamingContext, used for Spark Streaming applications. The SparkSession object introduced in Spark 2.0 combines all these objects into a single entry point that can be used for all Spark applications.

Through its SparkContext and SparkConf child objects, the SparkSession object contains all the runtime configuration properties set by the user, including configuration properties such as the Master, application name, number of Executors, and more. Figure 3.2 shows the SparkSession object and some of its configuration properties within a `pyspark` shell.



```
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 2.2.0
      /_/

Using Python version 3.5.2 (default, Nov 17 2016 17:05:23)
SparkSession available as 'spark'.
>>> myconfig = sc._conf
>>> for param in myconfig.getAll():
...     print(param)
...
('spark.app.id', 'app-20170819035801-0007')
('spark.driver.port', '43657')
('spark.sql.catalogImplementation', 'hive')
('spark.rdd.compress', 'True')
('spark.serializer.objectStreamReset', '100')
('spark.executor.id', 'driver')
('spark.submit.deployMode', 'client')
('spark.driver.host', '172.31.22.81')
('spark.master', 'spark://sparkmaster:7077')
('spark.app.name', 'PySparkShell')
>>>
```

Figure 3.2 SparkSession properties.

SparkSession Name

The object name for the SparkSession instance is arbitrary. By default, the SparkSession instantiation in the Spark interactive shells is named `spark`. For consistency, you always instantiate the SparkSession as `spark`; however, the name is up to the developer's discretion.

Listing 3.1 demonstrates how to create a SparkSession within a non-interactive Spark application, such as a program submitted using `spark-submit`.

Listing 3.1 **Creating a SparkSession**

```
from pyspark.sql import SparkSession
spark = SparkSession.builder \
    .master("spark://sparkmaster:7077") \
    .appName("My Spark Application") \
    .config("spark.submit.deployMode", "client") \
    .getOrCreate()
numlines = spark.sparkContext.textFile("file:///opt/spark/licenses") \
    .count()
print("The total number of lines is " + str(numlines))
```

Application Planning

One of the main functions of the Driver is to plan the application. The Driver takes the application processing input and plans the execution of the program. The Driver takes all the requested *transformations* (data manipulation operations) and *actions* (requests for output or prompts to execute programs) and creates a *directed acyclic graph* (*DAG*) of *nodes*, each representing a transformational or computational step.

> **Directed Acyclic Graph (DAG)**
>
> A DAG is a mathematical construct that is commonly used in computer science to represent dataflows and their dependencies. DAGs contain vertices, or nodes, and edges. Vertices in a dataflow context are steps in the process flow. Edges in a DAG connect vertices to one another in a directed orientation and in such a way that it is impossible to have circular references.

A Spark application DAG consists of *tasks* and *stages*. A task is the smallest unit of schedulable work in a Spark program. A stage is a set of tasks that can be run together. Stages are dependent upon one another; in other words, there are *stage dependencies*.

In a process scheduling sense, DAGs are not unique to Spark. For instance, they are used in other Big Data ecosystem projects, such as Tez, Drill, and Presto for scheduling. DAGs are fundamental to Spark, so it is worth being familiar with the concept.

Application Orchestration

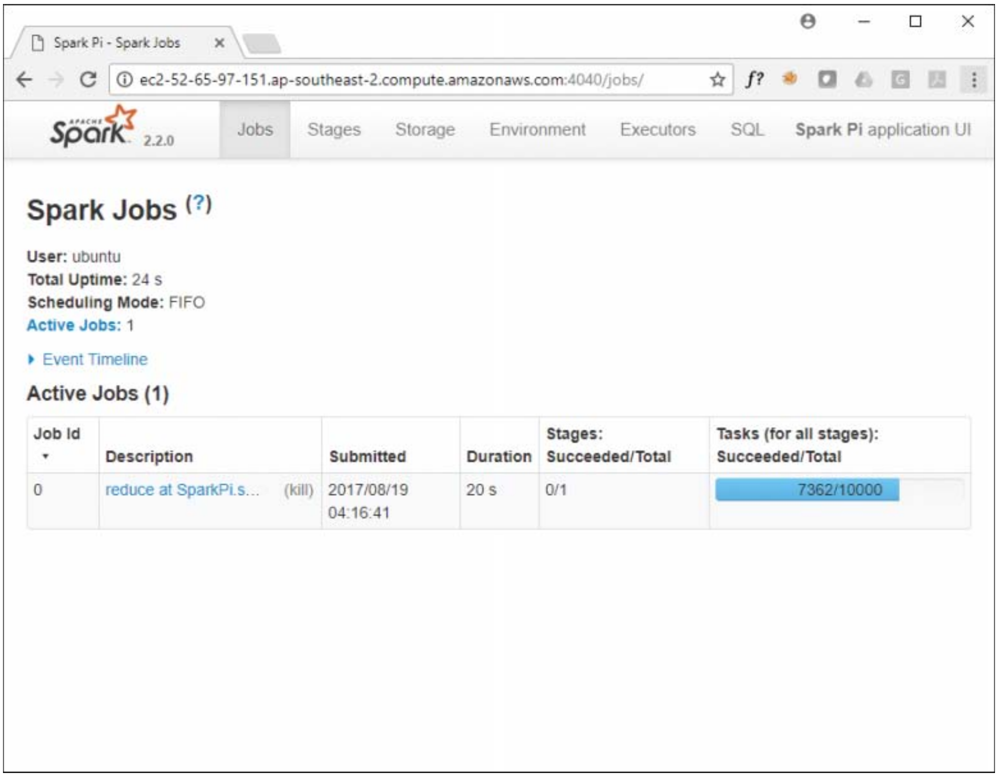
The Driver also coordinates the running of stages and tasks defined in the DAG. Key driver activities involved in the scheduling and running of tasks include the following:

- Keeping track of available resources to execute tasks
- Scheduling tasks to run “close” to the data where possible (the concept of data locality)

Other Functions

In addition to planning and orchestrating the execution of a Spark program, the Driver is also responsible for returning the results from an application. These could be return codes or data in the case of an action that requests data to be returned to the client (for example, an interactive query).

The Driver also serves the application UI on port 4040, as shown in Figure 3.3. This UI is created automatically; it is independent of the code submitted or how it was submitted (that is, interactive using `pyspark` or non-interactive using `spark-submit`).



Spark Jobs (?)

User: ubuntu
Total Uptime: 24 s
Scheduling Mode: FIFO
Active Jobs: 1

▸ Event Timeline

Active Jobs (1)

Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	reduce at SparkPi.s... (kill)	2017/08/19 04:16:41	20 s	0/1	7362/10000

Figure 3.3 Spark application UI.

If subsequent applications launch on the same host, successive ports are used for the application UI (for example, 4041, 4042, and so on).

Spark Workers and Executors

Spark Executors are the processes on which Spark DAG tasks run. Executors reserve CPU and memory resources on slave nodes, or Workers, in a Spark cluster. An Executor is dedicated to a specific Spark application and terminated when the application completes. A Spark program normally consists of many Executors, often working in parallel.

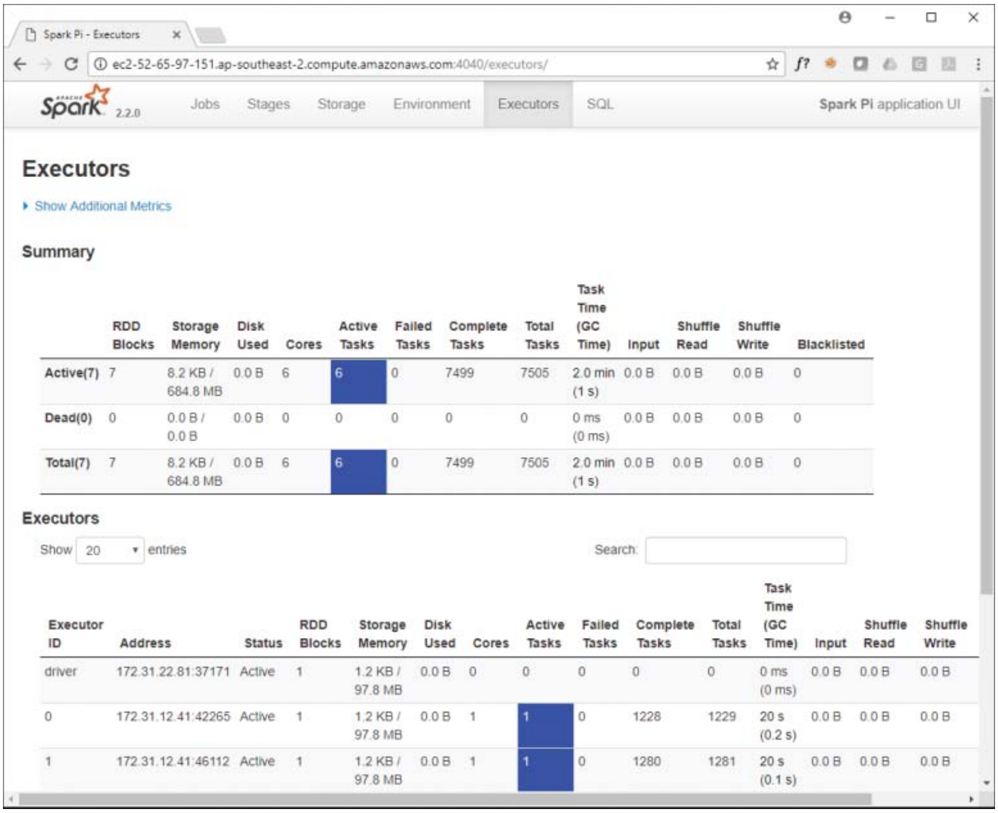
Typically, a Worker node—which hosts the Executor process—has a finite or fixed number of Executors allocated at any point in time. Therefore, a cluster—being a known number of nodes—has a finite number of Executors available to run at any given time. If an application requires Executors in excess of the physical capacity of the cluster, they are scheduled to start as other Executors complete and release their resources.

As mentioned earlier in this chapter, JVMs host Spark Executors. The JVM for an Executor is allocated a *heap*, which is a dedicated memory space in which to store and manage objects.

The amount of memory committed to the JVM heap for an Executor is set by the property `spark.executor.memory` or as the `--executor-memory` argument to the `pyspark`, `spark-shell`, or `spark-submit` commands.

Executors store output data from tasks in memory or on disk. It is important to note that Workers and Executors are aware only of the tasks allocated to them, whereas the Driver is responsible for understanding the complete set of tasks and the respective dependencies that comprise an application.

By using the Spark application UI on port 404*x* of the Driver host, you can inspect Executors for the application, as shown in Figure 3.4.



Executors

Summary

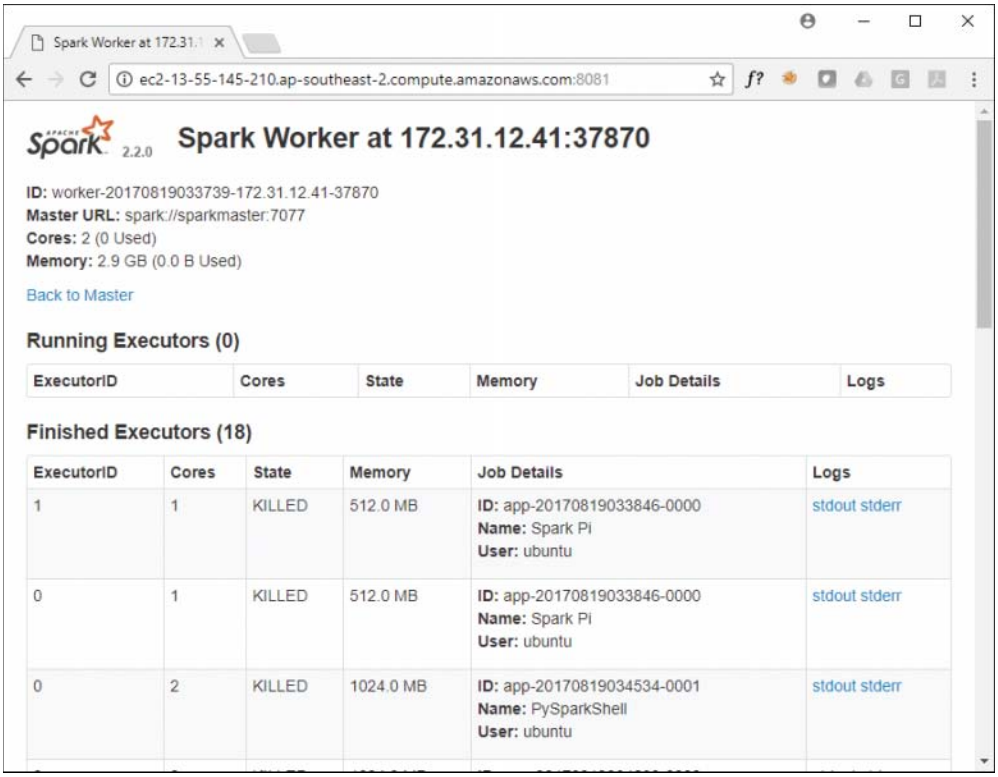
	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Blacklisted
Active(7)	7	8.2 KB / 684.8 MB	0.0 B	6	6	0	7499	7505	2.0 min (1 s)	0.0 B	0.0 B	0.0 B	0
Dead(0)	0	0.0 B / 0.0 B	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	0
Total(7)	7	8.2 KB / 684.8 MB	0.0 B	6	6	0	7499	7505	2.0 min (1 s)	0.0 B	0.0 B	0.0 B	0

Executors

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write
driver	172.31.22.81:37171	Active	1	1.2 KB / 97.8 MB	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B
0	172.31.12.41:42265	Active	1	1.2 KB / 97.8 MB	0.0 B	1	1	0	1228	1229	20 s (0.2 s)	0.0 B	0.0 B	0.0 B
1	172.31.12.41:46112	Active	1	1.2 KB / 97.8 MB	0.0 B	1	1	0	1280	1281	20 s (0.1 s)	0.0 B	0.0 B	0.0 B

Figure 3.4 Executors tab in the Spark application UI.

For Spark Standalone cluster deployments, a worker node exposes a user interface on port 8081, as shown in Figure 3.5.



Spark Worker at 172.31.12.41:37870

ec2-13-55-145-210.ap-southeast-2.compute.amazonaws.com:8081

2.2.0

ID: worker-20170819033739-172.31.12.41-37870
Master URL: spark://sparkmaster:7077
Cores: 2 (0 Used)
Memory: 2.9 GB (0.0 B Used)

Back to Master

Running Executors (0)

ExecutorID	Cores	State	Memory	Job Details	Logs

Finished Executors (18)

ExecutorID	Cores	State	Memory	Job Details	Logs
1	1	KILLED	512.0 MB	**ID:** app-20170819033846-0000 **Name:** Spark Pi **User:** ubuntu	stdout stderr
0	1	KILLED	512.0 MB	**ID:** app-20170819033846-0000 **Name:** Spark Pi **User:** ubuntu	stdout stderr
0	2	KILLED	1024.0 MB	**ID:** app-20170819034534-0001 **Name:** PySparkShell **User:** ubuntu	stdout stderr

Figure 3.5 Spark Worker UI.

The Spark Master and Cluster Manager

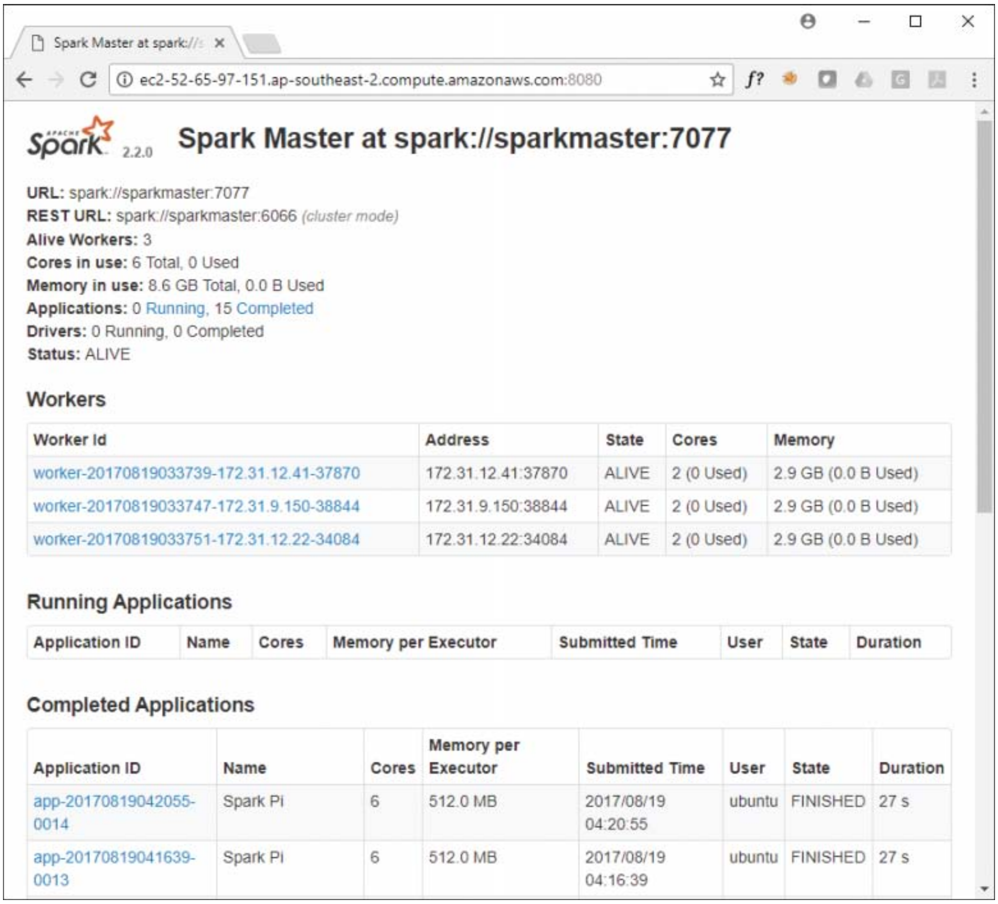
The Spark Driver plans and coordinates the set of tasks required to run a Spark application. The tasks themselves run in Executors, which are hosted on Worker nodes.

The Master and the Cluster Manager are the central processes that monitor, reserve, and allocate the distributed cluster resources (or containers, in the case of YARN or Mesos) on which the Executors run. The Master and the Cluster Manager can be separate processes, or they can combine into one process, as is the case when running Spark in Standalone mode.

Spark Master

The Spark Master is the process that requests resources in the cluster and makes them available to the Spark Driver. In all deployment modes, the Master negotiates resources or containers with Worker nodes or slave nodes and tracks their status and monitors their progress.

When running Spark in Standalone mode, the Spark Master process serves a web UI on port 8080 on the Master host, as shown in Figure 3.6.


Spark Master at spark://s
ec2-52-65-97-151.ap-southeast-2.compute.amazonaws.com:8080
Spark 2.2.0

Spark Master at spark://sparkmaster:7077

URL: spark://sparkmaster:7077
REST URL: spark://sparkmaster:6066 *(cluster mode)*
Alive Workers: 3
Cores in use: 6 Total, 0 Used
Memory in use: 8.6 GB Total, 0.0 B Used
Applications: 0 Running, 15 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers

Worker Id	Address	State	Cores	Memory
worker-20170819033739-172.31.12.41-37870	172.31.12.41:37870	ALIVE	2 (0 Used)	2.9 GB (0.0 B Used)
worker-20170819033747-172.31.9.150-38844	172.31.9.150:38844	ALIVE	2 (0 Used)	2.9 GB (0.0 B Used)
worker-20170819033751-172.31.12.22-34084	172.31.12.22:34084	ALIVE	2 (0 Used)	2.9 GB (0.0 B Used)

Running Applications

Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration

Completed Applications

Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration
app-20170819042055-0014	Spark Pi	6	512.0 MB	2017/08/19 04:20:55	ubuntu	FINISHED	27 s
app-20170819041639-0013	Spark Pi	6	512.0 MB	2017/08/19 04:16:39	ubuntu	FINISHED	27 s

Figure 3.6 Spark Master UI.

Spark Master Versus Spark Driver

It is important to distinguish the runtime functions of the Driver and the Master. The name *Master* may be inferred to mean that this process is governing the execution of the application—but this is not the case. The Master simply requests resources and makes those resources available to the Driver. Although the Master monitors the status and health of these resources, it is not involved in the execution of the application and the coordination of its tasks and stages. That is the job of the Driver.

Cluster Manager

The Cluster Manager is the process responsible for monitoring the Worker nodes and reserving resources on these nodes upon request by the Master. The Master then makes these cluster resources available to the Driver in the form of Executors.

As discussed earlier, the Cluster Manager can be separate from the Master process. This is the case when running Spark on Mesos or YARN. In the case of Spark running in Standalone mode, the Master process also performs the functions of the Cluster Manager. Effectively, it acts as its own Cluster Manager.

A good example of the Cluster Manager function is the YARN ResourceManager process for Spark applications running on Hadoop clusters. The ResourceManager schedules, allocates, and monitors the health of containers running on YARN NodeManagers. Spark applications then use these containers to host Executor processes, as well as the Master process if the application is running in `cluster` mode; we will look at this shortly.

Spark Applications Using the Standalone Scheduler

In Chapter 2, "Deploying Spark," you learned about the Standalone scheduler as a deployment option for Spark. You also deployed a fully functional multi-node Spark Standalone cluster in one of the exercises in Chapter 2. As discussed earlier, in a Spark cluster running in Standalone mode, the Spark Master process performs the Cluster Manager function as well, governing available resources on the cluster and granting them to the Master process for use in a Spark application.

Spark Applications Running on YARN

As discussed previously, Hadoop is a very popular and common deployment platform for Spark. Some industry pundits believe that Spark will soon supplant MapReduce as the primary processing platform for applications in Hadoop. Spark applications on YARN share the same runtime architecture but have some slight differences in implementation.

ResourceManager as the Cluster Manager

In contrast to the Standalone scheduler, the Cluster Manager in a YARN cluster is the YARN ResourceManager. The ResourceManager monitors resource usage and availability across all nodes in a cluster. Clients submit Spark applications to the YARN ResourceManager. The ResourceManager allocates the first container for the application, a special container called the *ApplicationMaster*.

ApplicationMaster as the Spark Master

The ApplicationMaster is the Spark Master process. As the Master process does in other cluster deployments, the ApplicationMaster negotiates resources between the application Driver and the Cluster Manager (or ResourceManager in this case); it then makes these resources (containers) available to the Driver for use as Executors to run tasks and store data for the application. The ApplicationMaster remains for the lifetime of the application.

Deployment Modes for Spark Applications Running on YARN

Two deployment modes can be used when submitting Spark applications to a YARN cluster: Client mode and Cluster mode. Let's look at them now.

Client Mode

In Client mode, the Driver process runs on the client submitting the application. It is essentially unmanaged; if the Driver host fails, the application fails. Client mode is supported for both interactive shell sessions (`pyspark`, `spark-shell`, and so on) and non-interactive application submission (`spark-submit`). Listing 3.2 shows how to start a `pyspark` session using the Client deployment mode.

Listing 3.2 **YARN Client Deployment Mode**

```
$SPARK_HOME/bin/pyspark \
--master yarn-client \
--num-executors 1 \
--driver-memory 512m \
--executor-memory 512m \
--executor-cores 1
# OR
$SPARK_HOME/bin/pyspark \
--master yarn \
--deploy-mode client \
--num-executors 1 \
--driver-memory 512m \
--executor-memory 512m \
--executor-cores 1
```

Figure 3.7 provides an overview of a Spark application running on YARN in Client mode.

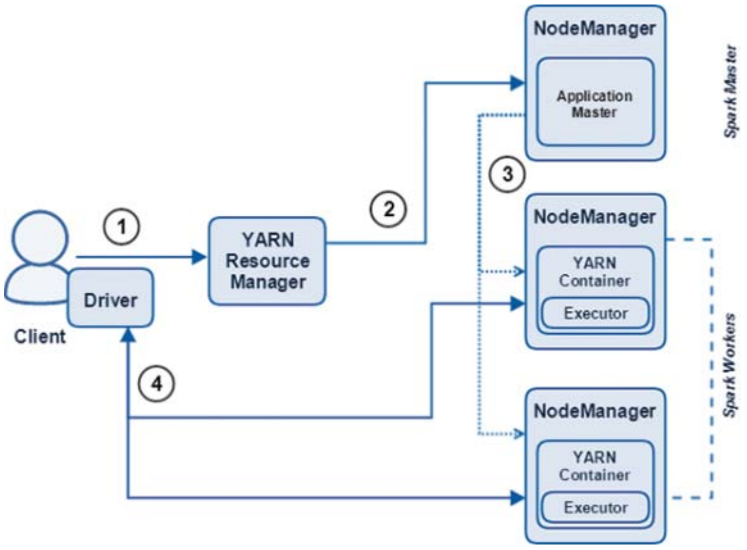


Figure 3.7 Spark application running in YARN Client mode.

The steps shown in Figure 3.7 are described here:

1. The client submits a Spark application to the Cluster Manager (the YARN ResourceManager). The Driver process, SparkSession, and SparkContext are created and run on the client.
2. The ResourceManager assigns an ApplicationMaster (the Spark Master) for the application.
3. The ApplicationMaster requests containers to be used for Executors from the ResourceManager. With the containers assigned, the Executors spawn.
4. The Driver, located on the client, then communicates with the Executors to marshal processing of tasks and stages of the Spark program. The Driver returns the progress, results, and status to the client.

The Client deployment mode is the simplest mode to use. However, it lacks the resiliency required for most production applications.

Cluster Mode

In contrast to the Client deployment mode, with a Spark application running in YARN Cluster mode, the Driver itself runs on the cluster as a subprocess of the ApplicationMaster. This provides resiliency: If the ApplicationMaster process hosting the Driver fails, it can be re-instantiated on another node in the cluster.

Listing 3.3 shows how to submit an application by using `spark-submit` and the YARN Cluster deployment mode. Because the Driver is an asynchronous process running in the cluster, Cluster mode is not supported for the interactive shell applications (`pyspark` and `spark-shell`).

Listing 3.3 **YARN Cluster Deployment Mode**

```
$SPARK_HOME/bin/spark-submit \
--master yarn-cluster \
--num-executors 1 \
--driver-memory 512m \
--executor-memory 512m \
--executor-cores 1
$SPARK_HOME/examples/src/main/python/pi.py 10000
# OR
$SPARK_HOME/bin/spark-submit \
--master yarn \
--deploy-mode cluster \
--num-executors 1 \
--driver-memory 512m \
--executor-memory 512m \
--executor-cores 1
$SPARK_HOME/examples/src/main/python/pi.py 10000
```

Figure 3.8 provides an overview of a Spark application running on YARN in Cluster mode.

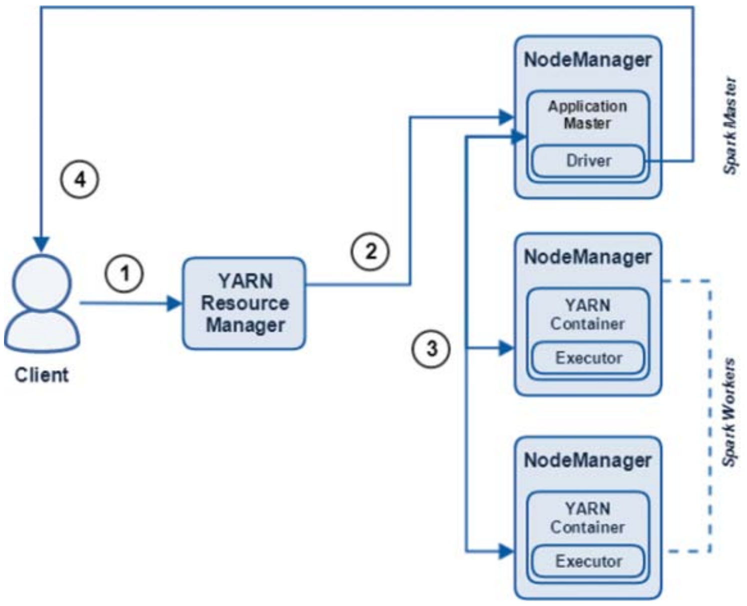


Figure 3.8 Spark application running in YARN Cluster mode.

The steps shown in Figure 3.8 are described here:

1. The client, a user process that invokes `spark-submit`, submits a Spark application to the Cluster Manager (the YARN ResourceManager).
2. The ResourceManager assigns an ApplicationMaster (the Spark Master) for the application. The Driver process is created on the same cluster node.
3. The ApplicationMaster requests containers for Executors from the ResourceManager. Executors are spawned within the containers allocated to the ApplicationMaster by the ResourceManager. The Driver then communicates with the Executors to marshal processing of tasks and stages of the Spark program.
4. The Driver, running on a node in the cluster, returns progress, results, and status to the client.

The Spark application web UI, as shown previously, is available from the ApplicationMaster host in the cluster; a link to this user interface is available from the YARN ResourceManager UI.

Local Mode Revisited

In Local mode, the Driver, the Master, and the Executor all run in a single JVM. As discussed earlier in this chapter, this is useful for development, unit testing, and debugging, but it has limited use for running production applications because it is not distributed and does not scale. Furthermore, failed tasks in a Spark application running in Local mode are not re-executed by default. You can override this behavior, however.

When running Spark in Local mode, the application UI is available at http://localhost:4040. The Master and Worker UIs are not available when running in Local mode.

Summary

In this chapter, you have learned about the Spark runtime application and cluster architecture, the components or a Spark application, and the functions of these components. The components of a Spark application include the Driver, Master, Cluster Manager, and Executors. The Driver is the process that the client interacts with when launching a Spark application, either through one of the interactive shells or through the `spark-submit` script. The Driver is responsible for creating the SparkSession object (the entry point for any Spark application) and planning an application by creating a DAG consisting of tasks and stages. The Driver communicates with a Master, which in turn communicates with a Cluster Manager to allocate application runtime resources (containers) on which Executors will run. Executors are specific to a given application and run all tasks for the application; they also store output data from completed tasks. Spark's runtime architecture is essentially the same regardless of the cluster resource scheduler used (Standalone, YARN, Mesos, and so on).

Now that we have explored Spark's cluster architecture, it's time to put the concepts into action starting in the next chapter.

This page intentionally left blank

Index

Symbols

<- (assignment) operator, 244
: (colon), 19
{ } (curly braces), 19
() (parentheses), 19
[ ] (square brackets), 19
0MQ (ZeroMQ), 228
7-Zip, 34

A

accumulator() method, 117
accumulators, 116–117
  accumulator() method, 117
  custom, 117–118
  DStreams, 216
  sample exercise, 119–120
  uses for, 118
  value() method, 117
actions, 48, 81
  collect(), 82, 128, 152
  count(), 81, 128
  defined, 59, 60
  example of, 72
  first(), 83
  fold(), 84–85
  foreach(), 85
  reduce(), 84
  take(), 82, 152
  takeSample(), 152
  top(), 82–83
ActiveMQ, 228
addInPlace() function, 118
aggregateByKey() function, 150
algorithmic functions, 184
allocation, dynamic, 153

ALS (Alternating Least Squares) technique
  Spark ML, 272–273
  Spark MLlib, 266
Amaterasu, 148
Amazon Kinesis, 237
  Analytics, 237
  documentation, 240
  Firehose, 237
  KCL (Kinesis Client Library), 238–239
  KPL (Kinesis Producer Library), 238
  Spark support for, 239
  Streams
    createStream() method, 239–240
    defined, 237–238
Amazon Redshift, 237
Amazon S3, 237
Amazon Software License (ASL), 239
Amazon SQS (Simple Queue Service), 228
Amazon Web Services. *See* **AWS (Amazon Web Services), Spark deployment on**
Ambari for Hortonworks, 148
Analytics, Amazon Kinesis, 237
Anderson, Edgar, 251
%angular (AngularJS) interpreter, 279
anonymous functions, 23–24
Apache Amaterasu, 148
Apache Cassandra, 201–204
Apache Hive, 8, 40
  accessing, 164
  CLI (command-line interface), 164
  data model
    complex types, 176
    primitive types, 175–176
  datatypes, 164–165
  HCatalog, 164
  HiveServer2, 164
  metastore, 163–164
  objects, 163–164
  overview of, 162
  tables
    creating DataFrames from, 170–171
    writing DataFrame output to, 188
Apache Kafka
  architecture, 229–230
  createDirectStream() method, 232–234
  KafkaUtils, 232
  sample application, 234–237
  Spark support for, 230–232
Apache Lucene, 6
Apache Mesos, 30
Apache Parquet
  file compression, 174
  overview of, 173
Apache Pig, 8, 40, 164
Apache Software Foundation (ASF), 6
Apache Solr, 206
Apache Zeppelin
  interpreters, 279
  notebooks, 278–279
Apache ZooKeeper, 230, 234–237
APIs (application programming interfaces)
  ConsumerConnector, 231
  SimpleConsumer, 231
append output mode, 226
ApplicationMasters
  overview of, 11–12
  as Spark Master, 53
applications
  application UI, 48–49
  architecture, 45–46
    Cluster Managers, 52–53
    Driver, 46–49
    Executors, 49–51
    illustrated, 46
    Masters, 51–52
    Standalone scheduler, 53
    Workers, 49–51
    YARN (Yet Another Resource Negotiator), 53–57
  Bay Area Bike Share exercise, 100–103
  checkpointing exercise, 136–138
  external
    accessing Spark SQL with, 194
    processing RDDs with, 138–139
  optimizing, 152–153
    dynamic allocation, 153
    parallelism, 152–153
    performance issues, 155–159
  orchestration, 48
  planning, 48
  scheduling, 10–13
  Spark Streaming exercise, 218–219
  Spark with Kafka exercise, 234–237
  WordCount exercise, 92–95

architecture
  Apache Kafka, 229–230
  Spark clusters, 45–46
    Cluster Managers, 52–53
    Driver, 46–48
    Executors, 49–51
    illustrated, 46
    Masters, 51–52
    Standalone scheduler, 53
    Workers, 49–51
    YARN (Yet Another Resource Negotiator), 53–57
  Spark SQL, 166–167
    extensions, 166, 167
    SparkSession entry point, 167–168
  Spark Streaming, 210–211
ARRAY datatype, 165
arrays
  Hive, 165
  R language, 245
  Spark primitive type, 176
ArrayType, 176
ASF (Apache Software Foundation), 6
ASL (Amazon Software License), 239
assignment operator (<-), 244
associative operations, optimizing, 149–150
average-word-length folder (GitHub), 120
Avro, 17, 229
awaitTermination() method, 211
AWS (Amazon Web Services), Spark deployment on, 39
  EC2 (Elastic Compute Cloud), 39
  EMR (Elastic MapReduce), 40–41
  GCP (Google Cloud Platform), 41

B

batch submissions, 16, 251
Bay Area Bike Share exercise, 100–103
Bayes' theorem, 266
beeline shell
  overview of, 193
  sample exercise, 194–195
Beeswax, 164
Berners-Lee, Tim, 161
Big Data, history of, 6–7. *See also* **Hadoop**
BIGINT datatype, 165
bin/ directory, 36
BINARY datatype, 165
BinaryType, 176
bisecting k-means, 273
bloom filters, 199
BOOLEAN datatype, 165
Boolean types
  DataFrame API, 176
  Hive, 165
BooleanType, 176
boto3 library, 205
broadcast() method, 112–113
broadcast variables, 112
  broadcast() method, 112–113
  DStreams, 216
  sample exercise, 119–120
  unpersist() method, 114–116
  value() method, 113
brokers (Kafka), 229
built-in DataFrame functions, 183–184
ByteType, 175
BZIP2 format, 62

C

c() function, 246
cache() method, 74, 187, 215
cacheTable() method, 187
caching
  DataFrames, 187
  DStreams, 215
  RDDs (Resilient Distributed Datasets)
    example of, 131
    when to use, 134
Cafarella, Mike, 6
call_func function, 25
cartesian() function, 99–100
Cassandra, 7, 201–204
Cassandra Query Language (CQL), 202
CDH (Cloudera Distribution of Hadoop), 32
cell_contents function, 25
cells (HBase), updating, 199
Character datatype, 245
character functions, 249
checkpoint() method, 135, 215
checkpointing, 134–135
  checkpoint() method, 135, 215
  DStreams, 214–215
  getCheckpointFile() method, 136

isCheckpointed() method, 136
sample exercise, 136–138
setCheckpointDir() method, 135
checkpointing folder (GitHub), 138
child RDDs (Resilient Distributed Datasets), 74–75
CLA (contributor license agreement), 6
classes. *See also* **objects**
CoGroupedRDD, 76
DataFrameReader, 224
DataFrames, 76
DoubleRDD, 76
HadoopRDD, 76
HashPartitioner, 121
JdbcRDD, 77
KafkaUtils, 232
NewHadoopRDD, 76
PairRDD, 76
PartitionPruningRDD, 77
SchemaRDD, 76
SequenceFileRDD, 76
ShuffledRDD, 77
SparkSession, 167–168
UnionRDD, 77
classification
decision trees, 262–266, 271–272
defined, 259–260
Naive Bayes, 266
Spark ML, 271–272
Spark MLlib, 262
clearCache() method, 187
Client deployment mode, 28–29, 53–55
Client Library (Kinesis), 238–239
closures, 24–25, 151–152
cloud, Spark deployment in, 39
AWS (Amazon Web Services), 39–41
Databricks, 42–43
Cloud Dataproc, 41
Cloudera Distribution of Hadoop (CDH), 32
Cloudera Manager, 148
cluster architecture, 8, 45–46
application clustering, 260–261
Cluster deployment mode, 28–29, 55–56, 143–144
Cluster Managers, 52–53
clustering keys, 202
Driver, 46–49
Executors, 49–51
illustrated, 46
k-means
Spark ML, 273–274
Spark MLlib, 269–270
Masters, 51–52
multi-node standalone clusters, 37–39
nodes, 8
Spark ML, 273–274
Standalone scheduler, 53
Workers, 49–51
YARN (Yet Another Resource Negotiator), 53–57
Cluster deployment mode, 28–29, 55–56, 143–144
Cluster Managers, 52–53, 142
cmp() method, 20
coalesce() method, 124, 154
coarse-grained transformations, 72
Coe, David Allan, 45
cogroup() function, 98–99
CoGroupedRDD, 76
collaborative filtering
defined, 260
Spark ML, 272–273
Spark MLlib, 266–267
collect() action, 82, 128, 152
collection of data, optimizing, 152
collections, 157–158, 202
colon (:), 19
columnar storage, 166, 173
columns() method, 179
combineByKey() function, 150
combiners, 150
commands
cqlsh, 202
easy_install, 200
java -version, 33, 34
library, 249
matrix, 247
pip, 200
pyspark, 30
R CMD INSTALL, 249
read, 248
sparkR, 250
spark-shell, 30
spark-submit, 16, 30, 55–56, 192
wget, 33

comma-separated value files. *See* **CSV (comma-separated value) files**
complete output mode, 227
complex types, 176
compression, file, 61
conf/ directory, 37
configuration, Spark. *See* **Spark configuration**
console sinks, 226
ConsumerConnector API, 231
consumers (Kafka), 229
containers, 11
contributor license agreement (CLA), 6
Core API. *See* **RDDs (Resilient Distributed Datasets)**
Couchbase, 206
CouchDB, 206
count() action, 81, 128
cPickle module, 22
CQL (Cassandra Query Language), 202
cqlsh utility, 202
CRAN, 249
CREATE TABLE statement, 165
createDataFrame() method, 169, 252
createDirectStream() method, 232–234
createRDD() method, 233
createStream() method, 239–240
cross joins, 99–100
CSV (comma-separated value) files
  creating data frames from, 252
  writing DataFrame output to, 188–191
csv() method, 189–190
curly braces ({}), 19
custom accumulators, 117–118
Cutting, Doug, 6

D

daemon environment variables, 144
DAGs (directed acyclic graphs), 48, 127
data collection, optimizing, 152
data/ directory, 37
data frames (R), 247–248. *See also* **DataFrames**
  creating, 251–253
  defined, 245
data locality, 7, 62–63
Data Manipulation Language (DML), 161
data mining, SparkR and, 254–255
data model (Hive)
  complex types, 176
  primitive types, 175–176
data sampling, 139
  sample() function, 140
  takeSample() function, 140–141
data science, with R language, 14, 244. *See also* **machine learning**
  data frames, creating, 247–248, 251–253
  data structures, 245–247
  datatypes, 245
  functions, 248–249
  packages, 248–249
  SparkR, 243
    accessing, 250–251
    data frames, 251–253
    data mining, 254–255
    documentation, 250
    linear regression, 255–256
    predictive analytics, 253–254
    predictive modeling, 254–255
    RStudio with, 257–258
data sinks, 225
  console sinks, 226
  file sinks, 225–226
  memory sinks, 226
data sources
  creating RDDs from, 66–69
  Structured Streaming, 224
    file sources, 224–225
    socket sources, 225
data structures
  Python
    dicts, 19–20
    lists, 18
    sets, 18
    tuples, 18–19
  R language, 245–247
Databricks, Spark deployment on, 42–43
Databricks File System (DBFS), 43
DataFrameReader, 224, 274
DataFrames, 76
  caching, 187
  complex types, 176
  converting to RDDs, 175
  creating
    from existing RDDs, 169

from flat files, 172–175
from Hive tables, 170–171
from JSON objects, 171–172
defined, 168–169
metadata, 179
multiple
grouping, 187
joining, 185–186
ordering, 186–187
operations
built-in functions, 183–184
cache(), 187
cacheTable(), 187
clearCache(), 187
columns(), 179
createDataFrame(), 169
csv(), 189–190
distinct(), 182–183
drop(), 181
dtypes(), 179
explain(), 183
filter(), 181
groupBy(), 187
intersect(), 186
join(), 185–186
json(), 171–172
orc(), 174–175
orderBy(), 186–187
parquet(), 173–174, 190–191
persist(), 187
printSchema(), 176
rdd(), 175
sample(), 183
sampleBy(), 183
saveAsTable(), 188
select(), 180–181
show(), 180
sql(), 170
subtract(), 186
table(), 170–171
text(), 173
udf(), 184–185
unpersist(), 187
output, saving
to files, 188–191
to Hive tables, 188
persistence, 187
primitive types, 175–176
repartitioning, 187
schemas
defining, 178
inferring, 176
DataFrameWriter, 274
DataNode process, 8–9
datasets. *See also* **RDDs (Resilient Distributed Datasets)**
datasets package, 251
golf/weather, 262–263
Movielens, 267–269
mtcars, 251–252
splitting, 263–264
DataStax Enterprise, 203
datatypes
DataFrame data model
complex types, 176
primitive types, 175–176
Hive, 164–165
R language, 245
DATE datatype, 165
date datatypes
Hive, 165
Spark primitive, 176
date functions, 184
DateType, 176
DBFS (Databricks File System), 43
dbutils library, 43
decision trees, 262–266, 271–272
DecisionTree.trainClassifier() function, 265
declarative referential integrity (DRI), 163
deep learning, 41
def keyword, 23
DEFLATE compression method, 62
DenseVector object, 271
dependencies, stage, 48
deployment, Spark. *See* **Spark deployment**
deployment modes
Client, 28–29, 53–55
Cluster, 28–29, 55–56
Local, 28, 56–57
on Mesos, 30
Spark Standalone, 28–29
diagrams, Venn, 103
dicts (dictionaries), 19–20

Direct Stream Access, 231
directed acyclic graphs (DAGs), 48, 127
directives, local, 28
directories, Spark installation, 36–37
disabling IPv6, 35
discretized streams. *See* **DStreams**
DISK_ONLY constant, 129, 130
DISK_ONLY_2 constant, 130
distinct() method, 79–80, 182–183
DML (Data Manipulation Language), 161
docstrings, 183
document stores, 197
documentation
  Amazon Kinesis, 240
  RDDs (Resilient Distributed Datasets), 77
  SparkR, 250
DOUBLE datatype, 165
double datatypes
  Hive, 165
  Spark primitive type, 176
DoubleRDD, 76
DoubleType, 176
downloading Spark, 31–32
DRI (declarative referential integrity), 163
Drivers, 46
  application orchestration, 48
  application planning, 48
  application UI, 48–49
  SparkSession, 46–48
drop() method, 181
DROP TABLE statement, 165
drop_duplicates() method, 182
DStreams, 211–212
  broadcast variables and accumulators, 216
  caching and persistence, 215
  lineage and checkpointing, 214–215
    DStream.checkpoint() method, 215
    StreamingContext.checkpoint() method, 215
  output operations, 216–218
    foreachRDD() method, 217–218
    pprint() method, 216
    saveAsTextFiles() method, 217
  sources, 212
    socketTextStream() method, 212–213
    textFileStream() method, 213
  transformations, 213–214
dtypes() method, 179
dynamic allocation, 153
DynamoDB, 204–206

E

easy_install command, 200
EC2 (Elastic Compute Cloud), Spark deployment on, 39
Edison, Thomas, 27
Elastic MapReduce. *See* **EMR (Elastic MapReduce)**
Elasticsearch, 206
EMR (Elastic MapReduce)
  GCP (Google Cloud Platform), 41
  Spark deployment on, 40–41
EMS (Enterprise Message Service),228
ensembles, 230
environment variables, 141–142
  defaults, 147
  HADOOP_CONF_DIR, 142
  HADOOP_HOME, 35, 142
  HIVE_CONF_DIR, 143
  JAVA_HOME, 142
  PYSPARK_DRIVER_PYTHON, 142
  PYSPARK_PYTHON, 142
  SPARK_CLASSPATH, 144
  SPARK_DAEMON_MEMORY, 144
  SPARK_DRIVER_MEMORY, 143
  SPARK_EXECUTOR_CORES, 143
  SPARK_EXECUTOR_INSTANCES, 143
  SPARK_EXECUTOR_MEMORY, 143
  SPARK_HOME, 33, 142
  SPARK_LOCAL_IP, 144
  SPARK_MASTER_IP, 144
  SPARK_MASTER_OPTS, 144
  SPARK_MASTER_PORT, 144
  SPARK_MASTER_WEBUI_PORT, 144
  SPARK_PUBLIC_DNS, 144
  SPARK_WORKER_CORES, 144
  SPARK_WORKER_DIR, 144
  SPARK_WORKER_INSTANCES, 144
  SPARK_WORKER_MEMORY, 144
  SPARK_WORKER_OPTS, 144
  SPARK_WORKER_PORT, 144
  SPARK_WORKER_WEBUI_PORT, 144
  SPARK_YARN_APP_NAME, 143
  SPARK_YARN_DIST_ARCHIVES, 143

SPARK_YARN_DIST_FILES, 143
SPARK_YARN_QUEUE, 143
SPARKR_DRIVER_R, 142
YARN_CONF_DIR, 142
errors, searching log files for, 61
Estimator objects, 274
evaluation, lazy, 73
event processing. *See* **Spark Streaming**
examples/ directory, 37
execution, lazy, 73
Executors (Spark), 49–51
explain() function, 183
extensions, Spark SQL, 166
external applications, accessing Spark SQL with, 194
external programs, processing RDDs with, 138
pipe() method, 138–139
potential problems with, 138
extraction, features, 261

F

fault tolerance, RDDs (Resilient Distributed Datasets), 76
features
defined, 261
extraction, 261
file sinks, 225–226
file sources (Structured Streaming), 224–225
file systems
DBFS (Databricks File System), 43
HDFS (Hadoop Distributed File System), 7–8. *See also* HBase
blocks, 8–9
as data source for Spark, 17
defined, 7–8
metadata, 9
processes, 8–9
read operations, 9–10
write operations, 9–10
schemes and URI structures, 63
FileNotFoundException, 63
files
compression, 61
creating data frames from, 172–175, 252
creating RDDs from, 61
data locality, 62–63
file compression, 61
JSON files, 69–70
object files, 66
text files, 63–66
hadoop.dll, 34
log, searching, 61
log4j.properties, 136
log4j.properties.erroronly, 136
looping_test.py, 137
ORC (Optimized Row Columnar), 173
saving RDDs as, 217
shakespeare.txt, 219
spark-defaults.conf, 38
spark-env.sh, 38
spark-streaming-kafka-assembly.jar file, 232
stop-word-list.csv, 119
winutils.exe, 34
writing DataFrame output to, 188–191
filter() function, 24, 79, 122, 153, 181
filtering
bloom filters, 199
collaborative
defined, 260
Spark ML, 272–273
Spark MLlib, 266–267
DataFrames, 181
optimization and, 149
fine-grained transformations, 72
Firehose, Amazon Kinesis, 237
first() action, 83
fit() method, 274
flags, storage-level, 129–130
flat files, DataFrames created from, 172–175
orc() method, 174–175
parquet() method, 173–174
text() method, 173
flatMap() function, 24, 78, 182
flatMapValues() function, 87–89
FLOAT datatype, 165
FloatType, 176
Flume, 8
fold() action, 84–85
foldByKey() function, 91, 150
folders, GitHub
average-word-length, 120
checkpointing, 138

joining-datasets, 103
recommendation-engine, 269
streaming-wordcount, 219
followers (Kafka), 230
foreach() action, 85
foreachPartition() method, 125–126
foreachRDD() method, 217–218
format() method, 224, 225
frame() function, 247–248
Franklin, Benjamin, 209
full outer joins
defined, 96
fullOuterJoin() transformation, 98
fullOuterJoin() function, 98
functions and methods
accumulator(), 117
addInPlace(), 118
aggregateByKey(), 150
anonymous, 23–24
awaitTermination(), 211
broadcast(), 112–113
built-in, 183–184
c(), 246
cache(), 74, 187, 215
cacheTable(), 187
call_func, 25
cartesian(), 99–100
cell_contents, 25
checkpoint(), 135, 215
clearCache(), 187
closures, 24–25
cmp(), 20
coalesce(), 124, 154
cogroup(), 98–99
collect(), 82, 128, 152
columns(), 179
combineByKey(), 150
count(), 81, 128
createDataFrame(), 169, 252
createDirectStream(), 232–234
createRDD(), 233
createStream(), 239–240
csv(), 189–190, 248
df(), 252
distinct(), 79–80, 182–183
drop(), 181
drop_duplicates(), 182
dtypes(), 179
explain(), 183
filter(), 24, 79, 122, 153, 181
first(), 83
fit(), 274
flatMap(), 24, 78, 182
flatMapValues(), 87–89
fold(), 84–85
foldByKey(), 91, 150
foreach(), 85
foreachRDD(), 217–218
format(), 224, 225
frame(), 247–248
fullOuterJoin(), 98
fwf(), 248
generate_message, 25
getCheckpointFile(), 136
getStorageLevel(), 130
glm(), 255
groupBy(), 80, 187
groupByKey(), 89, 149
hadoopFile(), 66
higher-order functions, 24
intersect(), 186
intersection(), 104
isCheckpointed(), 136
jdbc(), 68–69
join(), 96–97, 185–186
json(), 69–70, 171–172, 253
keyBy(), 86–87
keys(), 20, 86
lambda syntax, 23–24
leftOuterJoin(), 97
len(), 20
library(), 249
load(), 268
map(), 18, 24, 77–78, 182
mapValues(), 87
max(), 106
mean(), 106
min(), 105–106
named, 23–24
objectFile(), 66
optimizing, 151–152
orc(), 174–175
orderBy(), 186–187
outputMode(), 227

parallelize(), 71
parquet(), 173–174, 190–191, 253
partitionBy(), 123
persist(), 73–74, 132, 187, 215
pickleFile(), 22
pipe(), 138–139
pprint(), 216
predict(), 256, 266, 266
printSchema(), 176
quit(), 36
randomSplit(), 263–264
range(), 71–72
rdd(), 175
read(), 67
readStream(), 224
reduce(), 84
reduceByKey(), 24, 90, 149–150
reduceByKeyAndWindow(), 223
repartition(), 123–124, 154
repartitionAndSortWithinPartitions(), 124–125
ret_message(), 25
rightOuterJoin(), 97
sample(), 140, 183
sampleBy(), 183
save(), 268
saveAsPickleFile(), 22
saveAsTable(), 188
saveAsTextFile(), 17
saveAsTextFiles(), 217
select(), 180–181, 183
sequenceFile(), 66
session(), 253
setCheckpointDir(), 135
show(), 180
socketTextStream(), 212–213, 225
sortBy(), 81
sortByKey(), 91–92
sql(), 170, 253
start(), 211
stats(), 108
stdev(), 107
stop(), 211
subtract(), 104–105, 186
subtractByKey(), 105
sum(), 107
summary(), 255–256
table(), 170–171, 248
take(), 82, 152
takeSample(), 140–141, 152
text(), 173
textFile(), 17, 63–64
textFileStream(), 213
toDebugString(), 128
top(), 82–83
train(), 266
trainClassifier(), 265
transform(), 274
treeAggregate(), 150
treeReduce(), 150
tuple(), 19
udf(), 184–185
union(), 104
unpersist(), 114–116, 132–134, 187
updateStateByKey(), 220–221
value(), 113, 117
values(), 20, 86
variance(), 107
wholeTextFiles(), 64–66, 76
window(), 222–223
writeStream(), 227
zero(), 118

G

Gaussian mixture model (GMM), 273
GCP (Google Cloud Platform), Spark deployment on, 41
generate_message function, 25
getCheckpointFile() method, 136
getStorageLevel() function, 130
GitHub folders
average-word-length, 120
checkpointing, 138
joining-datasets, 103
recommendation-engine, 269
streaming-wordcount, 219
glm() function, 255
glom() method, 126
GMM (Gaussian mixture model), 273
golf dataset, 262–263
Google Cloud Platform (GCP), Spark deployment on, 41

Google whitepapers
- "The Google File System", 6
- "MapReduce: Simplified Data Processing on Large Clusters", 6, 13

graph stores, 197
groupBy() function, 80, 187
groupByKey() function, 89, 149
grouping DataFrames, 187
GZIP format, 62

H

HaaS (Hadoop-as-a-Service), 40
Hadoop, 7. *See also* **HBase**
- CDH (Cloudera Distribution of Hadoop), 32
- core components of, 7–8
- data locality, 7
- development of, 6–7
- "ecosystem" projects, 8
- environment variables, 142–143
- HaaS (Hadoop-as-a-Service), 40
- HDFS (Hadoop Distributed File System)
  - blocks, 8–9
  - as data source for Spark, 17
  - defined, 7–8
  - metadata, 9
  - processes, 8–10
  - read operations, 9–10
  - write operations, 9–10
- HDP (Hortonworks Data Platform), 32
- HUE (Hadoop User Experience), 164
- installation, 34
- MapReduce, 13
- schema-on-read system, 7
- shared nothing approach, 7
- YARN (Yet Another Resource Negotiator), 7–8
  - application scheduling with, 10–13
  - ApplicationMaster, 11–12
  - NodeManagers, 10–12
  - as resource scheduler for Spark, 17
  - ResourceManager, 10–12
  - Spark jobs, submitting, 30
  - Spark on, 28–29

HADOOP_CONF_DIR environment variable, 142
HADOOP_HOME environment variable, 35, 142
hadoop.dll, 34
hadoopFile() method, 66
HadoopRDD, 76
HappyBase Python package, 200
hashing functions, 184
HashPartitioner class, 121
HBase
- defined, 197
- HappyBase Python package, 200
- HFile objects, 199
- overview of, 7, 197–200
- sample exercise, 200–201
- Scala API, 200
- sparsity, 199
- tables, scanning, 198

HCatalog, 164
HDFS (Hadoop Distributed File System). *See also* **HBase**
- blocks, 8–9
- as data source for Spark, 17
- defined, 7–8
- metadata, 9
- processes
  - DataNode, 8–9
  - NameNode, 9
- read operations, 9–10
- write operations, 9–10

HDP (Hortonworks Data Platform), 32
HFile objects, 199
higher-order functions, 24
Hive
- accessing, 164
- CLI (command-line interface), 164
- data model
  - complex types, 176
  - primitive types, 175–176
- datatypes, 164–165
- HCatalog, 164
- HiveServer2, 164
- metastore, 163–164
- objects, 163–164
- overview of, 8, 40, 162
- tables
  - creating data frames from, 170–171, 253
  - writing DataFrame output to, 188

HIVE_CONF_DIR environment variable, 143
HiveContext, 47
Hopper, Grace Murray, 5

Hortonworks Data Platform (HDP), 32
HUE (Hadoop User Experience), 164

I

IaaS (Infrastructure-as-a-Service), 39
IaC (Infrastructure-as-Code), 39
IBM WebSphere MQ, 228
immutable lists. *See* **tuples**
indexes, secondary
  Apache Cassandra, 202
  DynamoDB, 204
inferring DataFrame schemas, 176
Infrastructure-as-a-Service (IaaS), 39
Infrastructure-as-Code (IaC), 39
ingestion, 8–9
initializing RDDs (Resilient Distributed Datasets), 61
inner joins, 96
Input Tables, 225
input/output types, 16
installation
  Hadoop, 34
  JDK (Java Development Kit)
    on Linux or Mac OS X, 33
    on Windows, 34
  Python, 34
  Spark. *See* Spark deployment
INT datatype, 165
Integer datatype, 245
IntegerType, 175
Interactive Computing Protocol, 277
interactive submission, 15
interpreters, Zeppelin, 279
intersect() function, 186
intersection() transformation, 104
IPv6, disabling, 35
IPython, 275–277
Iris Data dataset, 251
isCheckpointed() method, 136
items (DynamoDB), 204

J

jars/ directory, 37
Java, Spark support for, 14
Java Database Connectivity. *See* **JDBC (Java Database Connectivity)**
Java Development Kit. *See* **JDK (Java Development Kit), installing**
Java Message Service (JMS), 228
java -version command, 33, 34
Java Virtual Machine (JVM), 13, 46
JAVA_HOME environment variable, 142
JavaScript Object Notation. *See* **JSON (JavaScript Object Notation)**
JDBC (Java Database Connectivity), 43, 67
  JDBC/ODBC interface, 192
  JdbcRDD, 77
jdbc() method, 68–69
JDK (Java Development Kit), installing
  on Linux or Mac OS X, 33
  on Windows, 34
JMS (Java Message Service), 228
jobs (Spark), submitting
  in Local mode, 28
  to Mesos cluster, 30
  to standalone cluster, 29
  to YARN cluster, 30
join() function, 96–97, 185–186
join operations
  Bay Area Bike Share exercise, 100–103
  cartesian(), 99–100
  cogroup(), 98–99
  DataFrames, 185–186
  defined, 95
  fullOuterJoin(), 98
  join(), 96–97, 185–186
  leftOuterJoin(), 97
  optimizing, 97
  rightOuterJoin(), 97
  types of, 95–96
joining-datasets folder (GitHub), 103
JSON (JavaScript Object Notation), 20–21
  creating DataFrames from, 171–172
  files, creating RDDs from, 69–70
  json package, 20–21
  Jupyter (IPython) notebooks, 275–277
json() method, 69–70, 171–172
Jupyter (IPython), 275–277
JVM (Java Virtual Machine), 13, 46

K

Kafka
  architecture, 229–230
  createDirectStream() method, 232–234

KafkaUtils, 232
sample application, 234–237
Spark support for, 230–232
KafkaUtils class, 232
KCL (Kinesis Client Library), 238–239
kernels, Jupyter, 277
keyBy() function, 86–87
Keynes, John Maynard, 243
keys (Cassandra), 202
keys() function, 20, 86
keyspaces, 202
key/value stores, 19, 197
Kinesis, 237
Analytics, 237
documentation, 240
Firehose, 237
KCL (Kinesis Client Library), 238–239
KPL (Kinesis Producer Library), 238
Spark support for, 239
Streams
createStream() method, 239–240
defined, 237–238
k-means clustering
Spark ML, 273–274
Spark MLlib, 269–270
KMeans package, 270
KPL (Kinesis Producer Library), 238

L

LabeledPoint objects, 264–265
lambda syntax, 23–24
latent Dirichlet allocation (LDA), 273
lazy evaluation, 73
LDA (latent Dirichlet allocation), 273
learning
deep, 41
machine. *See* machine learning
supervised, 254
unsupervised, 254
left outer joins
defined, 96
leftOuterJoin() transformation, 97
leftOuterJoin() function, 97
len() method, 20
levels, storage
choosing, 131
table of, 128–129
libraries
boto3, 205
dbutils, 43
KCL (Kinesis Client Library), 238–239
KPL (Kinesis Producer Library), 238
LIBSVM (library for support vector machines), 274
NumPy, 264
Pandas, 264
R, 249
library() function, 249
LIBSVM (library for support vector machines), 274
licenses, contributor, 6
licenses/ directory, 37
lineage
DStreams, 214–215
DStream.checkpoint() method, 215
StreamingContext.checkpoint() method, 215
RDDs (Resilient Distributed Datasets), 74–75, 127–128
linear regression, SparkR and, 255–256
Linux, Spark installation on, 32–34
lists, 18, 19
load() function, 268
loading data into RDDs (Resilient Distributed Datasets), 61
Local deployment mode, 28, 56–57
local directive, 28
locality, data, 7, 62–63
log files, searching for errors, 61
log4j.properties file, 136
log4j.properties.erroronly file, 136
Logical datatype, 245
LongType, 175
longwords.collect() action, 128
longwords.count() action, 128
lookup() method, 126
looping_test.py file, 137
Lucene, 6

M

Mac OS X, Spark installation on, 32–34
machine learning, 259
classification
decision trees, 262–266, 271–272
defined, 259–260

Naive Bayes, 266
Spark ML, 271–273
Spark MLlib, 262
clustering, 260–261
k-means, 269–270, 273–274
Spark ML, 273–274
collaborative filtering
defined, 260
Spark MLlib, 266–267
feature extraction, 261
pipelines, 274–275
Spark ML
classification, 271–272
clustering, 273–274
collaborative filtering, 272–273
pipelines, 274–275
Spark MLlib
classification, 262
clustering, 269–270
collaborative filtering, 266–267
movie recommender application, 266–267
MAP datatype, 165
map() function, 18, 24, 77–78, 182
mapPartitions() method, 126–127
MapR Converged Data Platform, 32
MapReduce
GCP (Google Cloud Platform), 41
Spark deployment on, 40–41
WordCount exercise, 92–95
"MapReduce: Simplified Data Processing on Large Clusters" (whitepaper), 6, 13
MapType, 176
mapValues() function, 87
Masters, 38, 51–52
master/slave model, 8
math functions, 184
matrices, 245–247
matrix command, 247
matrix factorization, 266
MatrixFactorizationModel.load() function, 268
max() transformation, 106
%md (Markdown) interpreter, 279
mean() transformation, 106
Memcached, 206
memory sinks, 226
MEMORY_AND_DISK constant, 129, 130
MEMORY_AND_DISK_2 constant, 130
MEMORY_AND_DISK_SER constant, 130
MEMORY_AND_DISK_SER* constant, 129
MEMORY_AND_DISK_SER_2 constant, 130
MEMORY_ONLY constant, 129, 130
MEMORY_ONLY_2 constant, 130
MEMORY_ONLY_SER constant, 130
MEMORY_ONLY_SER* constant, 129
MEMORY_ONLY_SER_2 constant, 130
Mesos, 30
message-oriented middleware (MOM), 228
messaging systems, Spark with, 228
Amazon Kinesis, 237
Analytics, 237
createStream() method, 239–240
documentation, 240
Firehose, 237
KCL (Kinesis Client Library), 238–239
Kinesis Streams, 237–238
KPL (Kinesis Producer Library), 238
Spark support for, 239
Apache Kafka
architecture, 229–230
createDirectStream() method, 232–234
KafkaUtils, 232
sample application, 234–237
Spark support for, 230–232
MOM (message-oriented middleware), 228
metadata, 9
columns() method, 179
dtypes() method, 179
operations, 179
metastores
configuration, 35
Hive, 163–164
methods. *See* **functions and methods**
Microsoft Message Queuing (MSMQ), 228
min() transformation, 105–106
MLlib
classification, 262
clustering, 269–270
collaborative filtering, 266–267
movie recommender application, 266–267
model.save() function, 268
modules
cPickle, 22
pickle, 22
urllib2, 119

MOM (message-oriented middleware), 228
MongoDB, 206
movie recommender application, 266–267
Movielens dataset, 267–269
MSMQ (Microsoft Message Queuing), 228
mtcars dataset, 251–252
multi-node standalone clusters, 37–39
multiple DataFrames
  grouping, 187
  joining, 185–186
  ordering, 186–187

N

Naive Bayes, 266
NaiveBayes package, 266
named functions, 23–24
NameNode process, 9
names, SparkSession, 47
narrow operations, 74
neural networks, 41
NewHadoopRDD, 76
NodeManagers, 10–12
nodes, 8
  DAGs (directed acyclic graphs) of, 48
  quorum of, 230
non-interactive submission, 16
non-splittable compression formats, 62
NoSQL systems, 7, 195–196
  Apache Cassandra, 201–204
  characteristics of, 196
  DynamoDB, 204–206
  HBase
    defined, 197
    HappyBase Python package, 200
    overview of, 197–200
    sample exercise, 200–201
    Scala API, 200
  types of, 196–197
notebooks, 275
  Apache Zeppelin, 278–279
  Jupyter (IPython), 275–277
Numeric datatype, 245
numeric value operations, 105–106, 249
  max(), 106
  mean(), 106
  min(), 105–106
  stats(), 108
  stdev(), 107
  sum(), 107
  variance(), 107
NumPy, 264
Nutch, 6

O

object files
  creating RDDs from, 66
  defined, 66
objectFile() method, 66
objects. *See also* **classes; DataFrames**
  DenseVector, 271
  DStreams, 211–212
    broadcast variables and accumulators, 216
    caching and persistence, 215
    lineage and checkpointing, 214–215
    output operations, 216–218
    sources, 212–213
    transformations, 213–214
  Estimator, 274
  HFile, 199
  Hive, 163–164
  HiveContext, 47
  JSON (JavaScript Object Notation), creating DataFrames from, 171–172
  LabeledPoint, 264–265
  Pipeline, 274
  Row, 271
  SparkConf, 47
  SparkContext, 47, 211
  SparkSession, 46–48, 211, 250
  SparseVector, 271
  SQLContext, 47
  StreamingContext, 47, 211
  StructField, 178
  Transformer, 274
observations (R), 252
OFF_HEAP constant, 130
OFF_HEAP storage level, 129
operations. *See also* **functions and methods**
  narrow, 74
  wide, 75
operators, assignment (<-), 244
Optimized Row Columnar (ORC) files, 173

optimizing Spark, 148
- applications, 152–153
  - dynamic allocation, 153
  - parallelism, 152–153
  - performance issues, 155–159
- associative operations, 149–150
- data collection, 152
- filtering, 149
- functions and closures, 151–152
- join operations, 97
- partitions, 153–155

ORC (Optimized Row Columnar) files, 173
orc() method, 174–175
ORCFile format, Spark support for, 17
orderBy() function, 186–187
outer joins
- defined, 96
- transformations
  - fullOuterJoin(), 98
  - leftOuterJoin(), 97
  - rightOuterJoin(), 97

output modes (Spark Streaming), 226–227
output operations, DStreams, 216–218
- foreachRDD() method, 217–218
- pprint() method, 216
- saveAsTextFiles() method, 217

output sinks, 225
- console sinks, 226
- file sinks, 225–226
- memory sinks, 226

outputMode() method, 227

P

PaaS (Platform-as-a-Service), 39
packages
- datasets, 251
- defined, 20–21
- HappyBase, 200
- json, 20–21
- KMeans, 270
- NaiveBayes, 266
- pyspark-cassandra, 203
- R language, 248–249

PairRDDs
- defined, 76
- transformations, 85–92
  - flatMapValues(), 87–89
  - foldByKey(), 91
  - groupByKey(), 89
  - keyBy(), 86–87
  - keys(), 86
  - mapValues(), 87
  - reduceByKey(), 90
  - sortByKey(), 91–92
  - values(), 86

Pandas, 264
ParallelCollectionRDD, 77
parallelism, optimizing, 152–153
parallelize() method, 71
parent RDDs (Resilient Distributed Datasets), 74–75
parentheses (), 19
Parquet
- file compression, 174
- overview of, 173

parquet() method, 173–174, 190–191
Partial DAG execution (PDE), 166
partitionBy() function, 123
PartitionPruningRDD, 77
partitions, 120
- Apache Kafka, 229
- API methods
  - foreachPartition(), 125–126
  - glom(), 126
  - lookup(), 126
  - mapPartitions(), 126–127
- controlling, 121–122
- keys, 202
- optimal number of, 123
- optimizing, 153–155
- overview of, 120–121
- repartitioning functions
  - coalesce(), 124
  - partitionBy(), 123
  - repartition(), 123–124
  - repartitionAndSortWithinPartitions(), 124–125
- statistics, 166

PDE (Partial DAG execution), 166
persist() method, 73–74, 132, 187, 215
persistence
- DataFrames, 187
- DStreams, 215
- RDDs (Resilient Distributed Datasets), 73–74, 131–132
  - persist() method, 132
  - unpersist() method, 132–134
  - when to use, 134

Pi Estimator, 33, 38
Pickle, 22
pickleFile() method, 22
Pig, 8, 40, 164
pip command, 200
pipe() method, 138–139
pipelines, Spark ML, 274–275
planning applications, 48
Platform-as-a-Service (PaaS), 39
populating RDDs (Resilient Distributed Datasets), 61
pprint() method, 216
precedence, Spark configuration properties, 148
predict() function, 256, 266, 266
predictive analytics, SparkR and, 253–254
predictive modeling, SparkR and, 254–255
Presto, 40
primary keys, 202
primitive types, 175–176
printSchema() method, 176
probability functions, 249
processes
  DataNode, 8–9
  NameNode, 9
Producer Library (Kinesis), 238
producers (Kafka), 229
programming (Spark)
  PySpark shell, 15
  RDD (Resilient Distributed Dataset), 16
  Scala shell, 15
  Standalone scheduler
    multi-node standalone clusters, 37–39
    Standalone deployment mode, 28–29
  submission types
    interactive, 15
    non-interactive, 16
  Workers, 38
programming interfaces, 14
properties
  RDDs (Resilient Distributed Datasets), 60
  Spark configuration
    configuration management, 148
    precedence, 148
    setting, 145–147
    table of, 145
pyspark command, 30
PySpark shell, 15, 53–54
PYSPARK_DRIVER_PYTHON environment variable, 142
PYSPARK_PYTHON environment variable, 142
pyspark-cassandra package, 203
pyspark.mllib.clustering.KMeans package.270
Python, 17. *See also* **functions and methods**
  data structures
    dicts, 19–20
    lists, 18
    sets, 18
    tuples, 18–19
  docstrings, 183
  functions
    anonymous functions, 23–24
    closures, 24–25
    higher-order functions, 24
    lambda syntax, 23–24
    named functions, 23–24
  HappyBase package, 200
  installation, 34
  libraries
    boto3, 205
    NumPy, 264
    Pandas, 264
  modules
    cPickle, 22
    pickle, 22
    urllib2, 119
  PySpark shell, 15, 53–54
  serialization
    JSON (JavaScript Object Notation), 20–21
    Pickle, 22
python/ directory, 37
%python interpreter, 279
PythonRDD, 77

Q

quit() method, 36
quorum of nodes, 230

R

R CMD INSTALL command, 249
R/ directory, 37

R language, 244. *See also* **functions and methods; methods**
batch mode, 251
data frames
creating, 247–248, 251–253
defined, 245
data structures, 245–247
datasets
golf/weather, 262–263
Movielens, 267–269
mtcars, 251–252
splitting, 263–264
datatypes, 245
functions, 248–249
history of, 244
packages, 248–249, 251
SparkR, 243
accessing, 250–251
data frames, 251–253
data mining, 254–255
documentation, 250
linear regression, 255–256
predictive analytics, 253–254
predictive modeling, 254–255
RStudio with, 257–258
RabbitMQ, 228
randomSplit() function, 263–264
range() method, 71–72
rdd() method, 175
RDDs (Resilient Distributed Datasets), 111
actions, 81
collect(), 82
count(), 81
defined, 59, 60
example of, 72
first(), 83
fold(), 84–85
foreach(), 85
reduce(), 84
take(), 82
top(), 82–83
caching
example of, 131
when to use, 134
checkpointing, 134–135
checkpoint() method, 135
getCheckpointFile() method, 136
isCheckpointed() method, 136
sample exercise, 136–138
setCheckpointDir() method, 135
converting DataFrames to, 175
creating
from data sources, 66–69
from files, 61–63
from JSON files, 69–70
from object files, 66
programmatically, 71–72
from text files, 63–66
data sampling, 139
sample() function, 140
takeSample() function, 140–141
DataFrames created from, 169
defined, 16
documentation for, 77
explained, 59–61
fault tolerance, 76
join operations
Bay Area Bike Share exercise, 100–103
cartesian(), 99–100
cogroup(), 98–99
defined, 95
fullOuterJoin(), 98
join(), 96–97
leftOuterJoin(), 97
optimizing, 97
rightOuterJoin(), 98
types of, 95–96
lazy evaluation, 73
lineage, 74–75, 127–128
loading data into, 61
MapReduce
GCP (Google Cloud Platform), 41
Spark deployment on, 40–41
WordCount exercise, 92–95
numeric value operations, 105–106
max(), 106
mean(), 106
min(), 105–106
stats(), 108
stdev(), 107
sum(), 107
variance(), 107
PairRDDs
defined, 76
transformations, 85–92

parent/child, 74–75
partitions, 120
  controlling, 121–122
  foreachPartition() method, 125–126
  glom() method, 126
  lookup() method, 126
  mapPartitions() method, 126–127
  optimal number of, 123
  optimizing, 153–155
  overview of, 120–121
  statistics, 166
performing functions on, 217–218
persistence, 73–74, 131–132
  persist() method, 132
  unpersist() method, 132–134
  when to use, 134
processing with external programs, 138
  pipe() method, 138–139
  potential problems with, 138
properties of, 60
repartitioning functions
  coalesce(), 124
  partitionBy(), 123
  repartition(), 123–124
  repartitionAndSortWithinPartitions(), 124–125
reuse, 73–74
saving as text files, 217
set operations, 103
  intersection(), 104
  subtract(), 104–105
  subtractByKey(), 105
  union(), 104
shared variables
  accumulators, 116–119
  broadcast variables, 112–116
  sample exercise, 119–120
storage options
  getStorageLevel() function, 130
  storage levels, 128–129
  storage-level flags, 129–130
transformations, 77
  cartesian(), 99–100
  coarse-grained, 72
  cogroup(), 98–99
  defined, 59
  distinct(), 79–80
  example of, 72
  filter(), 79
  fine-grained, 72
  flatMap(), 78
  flatMapValues(), 87–89
  foldByKey(), 91
  fullOuterJoin(), 98
  groupBy(), 80
  groupByKey(), 89
  intersection(), 104
  join(), 96–97
  keyBy(), 86–87
  keys(), 86
  leftOuterJoin(), 97
  map(), 77–78
  mapValues(), 87
  max(), 106
  mean(), 106
  min(), 105–106
  reduceByKey(), 90
  rightOuterJoin(), 97
  sortBy(), 81
  sortByKey(), 91–92
  stats(), 108
  stdev(), 107
  subtract(), 104–105
  subtractByKey(), 105
  sum(), 107
  union(), 104
  values(), 86
  variance(), 107
types of, 76–77
read command, 248
  read.csv(), 248
  read.df(), 252
  read.fwf(), 248
  read.jdbc(), 68–69
  read.json(), 69–70, 171–172, 253
  read.parquet(), 253
  read.table(), 248
read() method, 67
read operations (HDFS), 9–10
readStream() method, 224
receivers (Kafka), 231
recommendation-engine folder (GitHub), 269
recommender application, 266–267
Redis, 206

Redshift, 237
reduce() action, 84
reduceByKey() function, 24, 90, 149–150
reduceByKeyAndWindow() method, 223
reflection, 176
region servers, 199
regions, 199
regression, linear, 255–256
repartition() method, 123–124, 154
repartitionAndSortWithinPartitions() method, 124–125
repartitioning
  DataFrames, 187
  RDDs (Resilient Distributed Datasets)
    coalesce(), 124
    partitionBy(), 123
    repartition(), 123–124
    repartitionAndSortWithinPartitions(), 124–125
Resilient Distributed Datasets. *See* **RDDs (Resilient Distributed Datasets)**
ResourceManagers, 10–12, 53
Result Tables, 225
ret_message() function, 25
Riak, 206
right outer joins
  defined, 96
  rightOuterJoin() transformation, 97
rightOuterJoin() function, 97
Row objects, 271
RStudio, SparkR and, 257–258

S

S3, 237
SaaS (Software-as-a-Service), 39
sample() function, 140, 183
sampleBy() function, 183
sampling data, 139
  sample() function, 140
  takeSample() function, 140–141
save() function, 268
saveAsPickleFile() method, 22
saveAsTable() method, 188
saveAsTextFile() function, 17
saveAsTextFiles() method, 217
saving DataFrame output
  to files, 188–191
  to Hive tables, 188
sbin/ directory, 37
Scala, 13, 14, 15, 31, 200
scanning HBase tables, 198
scheduler. *See* **Standalone scheduler**
scheduling applications, 10–13
schema-on-read systems, 7
schema-on-write systems, 7
SchemaRDD, 76
schemas, DataFrame, 252–253
  defining, 178
  inferring, 176
schemes, filesystem, 63
Scikit-learn project, 274
searching, log files for errors, 61
secondary indexes
  Apache Cassandra, 202
  DynamoDB, 204
select() method, 180–181, 183
sequenceFile() method, 66
SequenceFileRDD, 76
serialization
  JSON (JavaScript Object Notation), 20–21
  Pickle, 22
servers
  region servers, 199
  Spark History Server, 158–159
  Thrift JDBC/ODBC, 192, 194–195
session() function, 253
set operations, 103
  intersection(), 104
  subtract(), 104–105
  subtractByKey(), 105
  union(), 104
setCheckpointDir() method, 135
%sh (Shell commands) interpreter, 279
Shakespeare text-streaming application, 218–219
shared nothing operations, 7, 77
shared variables, 111
  accumulators, 116–117
    accumulator() method, 117
    custom, 117–118
    uses for, 118
    value() method, 117
  broadcast variables, 112
    broadcast() method, 112–113
    unpersist() method, 114–116
    value() method, 113
  sample exercise, 119–120

shells
beeline
overview of, 193
sample exercise, 194–195
PySpark, 15, 53–54
Scala, 15
sparkR, 250
spark-sql, 191
ShortType, 175
show() method, 180
ShuffledRDD, 77
shuffling, 73, 156–157
Simple Queue Service (SQS), 228
SimpleConsumer API, 231
sinks, data, 225
console sinks, 226
file sinks, 225–226
memory sinks, 226
sliding window operations, 221
reduceByKeyAndWindow() method, 223
window() method, 222–223
SMALLINT datatype, 165
Snappy, 62
socket sources (Structured Streaming), 225
socketTextStream() method, 212–213, 225
Software-as-a-Service (SaaS), 39
Solr, 206
sortBy() function, 81
sortByKey() function, 91–92
sources
DStream, 212, 213
socketTextStream() method, 212–213
textFileStream() method, 213
Structured Streaming, 224
file sources, 224–225
socket sources, 225
Spark, overview of, 13, 257–258
Hadoop and
HDFS (Hadoop Distributed File System), 17
YARN (Yet Another Resource Negotiator), 17
history of, 13
input/output types, 16
programming interfaces, 14
uses for, 14
Spark cluster architecture, 45–46
Cluster Managers, 52–53
Driver, 46–49
Executors, 49–51
illustrated, 46
Masters, 51–52
Standalone scheduler, 53
Workers, 49–51
YARN (Yet Another Resource Negotiator), 53–57
Spark configuration
environment variables, 141–142
defaults, 147
HADOOP_CONF_DIR, 142
HADOOP_HOME, 142
HIVE_CONF_DIR, 143
JAVA_HOME, 142
PYSPARK_DRIVER_PYTHON, 142
PYSPARK_PYTHON, 142
SPARK_CLASSPATH, 144
SPARK_DAEMON_MEMORY, 144
SPARK_DRIVER_MEMORY, 143
SPARK_EXECUTOR_CORES, 143
SPARK_EXECUTOR_INSTANCES, 143
SPARK_EXECUTOR_MEMORY, 143
SPARK_HOME, 142
SPARK_LOCAL_IP, 144
SPARK_MASTER_IP, 144
SPARK_MASTER_OPTS, 144
SPARK_MASTER_PORT, 144
SPARK_MASTER_WEBUI_PORT, 144
SPARK_PUBLIC_DNS, 144
SPARK_WORKER_CORES, 144
SPARK_WORKER_DIR, 144
SPARK_WORKER_INSTANCES, 144
SPARK_WORKER_MEMORY, 144
SPARK_WORKER_OPTS, 144
SPARK_WORKER_PORT, 144
SPARK_WORKER_WEBUI_PORT, 144
SPARK_YARN_APP_NAME, 143
SPARK_YARN_DIST_ARCHIVES, 143
SPARK_YARN_DIST_FILES, 143
SPARK_YARN_QUEUE, 143
SPARKR_DRIVER_R, 142
YARN_CONF_DIR, 142
optimizing, 148
applications, 152–153, 155–159
associative operations, 149–150
data collection, 152
filtering, 149
functions and closures, 151–152
partitions, 153–155

properties
  configuration management, 148
  precedence, 148
  setting, 145–147
  table of, 145

Spark Core API. *See* **RDDs (Resilient Distributed Datasets)**

Spark deployment, 27
in the cloud, 39
  AWS (Amazon Web Services), 39–41
  Databricks, 42–43
installation directory contents, 36–37
on Linux or Mac OS X, 32–34
Masters, 38
modes
  Client, 28–29, 53–55
  Cluster, 28–29, 55–56
  Local, 28, 56–57
  on Mesos, 30
  Spark Standalone, 28–29
multi-node standalone clusters, 37–39
preparation for, 30–31
releases, downloading, 31–32
requirements for, 31
on Windows, 34–36

Spark History Server, 158–159

%spark interpreter, 279

Spark ML
classification, 271–272
clustering, 273–274
collaborative filtering, 272–273
pipelines, 274–275

Spark MLlib
classification, 262
clustering, 269–270
collaborative filtering, 266–267
movie recommender application, 266–267

Spark SQL, 161. *See also* **DataFrames**
accessing, 191
  beeline shell, 193, 194–195
  external applications, 194
  sample exercise, 194–195
  spark-sql shell, 191
  Thrift JDBC/ODBC server, 192, 194–195
architecture, 166–167
  extensions, 166
  high-level architecture, 167
  SparkSession entry point, 167–168
HBase
  HFile objects, 199
  sparsity, 199
Hive
  accessing, 164
  CLI (command-line interface), 164
  datatypes, 164–165
  HCatalog, 164
  HiveServer2, 164
  metastore, 163–164
  objects, 163–164
  overview of, 162
  writing DataFrame output to, 188
NoSQL systems, 195–196
  Apache Cassandra, 201–204
  characteristics of, 196
  DynamoDB, 204–206
  HBase, 196–201
  types of, 196–197
reflection, 176

Spark Streaming, 209–210
architecture, 210–211
DataFrames
  basic operations, 180–183
  metadata operations, 179
DStreams, 211–212
  broadcast variables and accumulators, 216
  caching and persistence, 215
  lineage and checkpointing, 214–215
  output operations, 216–218
  sources, 212–213
  transformations, 213–214
goals of, 210
messaging systems, 228
  Amazon Kinesis, 237–240
  Apache Kafka, 229–237
output modes, 226–227
sample application, 218–219
sliding window operations, 221
  reduceByKeyAndWindow() method, 223
  window() method, 222–223
state operations, 219–221
Structured Streaming, 223–224
  data sinks, 225–226
  data sources, 224–225
structured streaming operations, 227–228

SPARK_CLASSPATH environment variable, 144
SPARK_DAEMON_MEMORY environment variable, 144
SPARK_DRIVER_MEMORY environment variable, 143
SPARK_EXECUTOR_CORES environment variable, 143
SPARK_EXECUTOR_INSTANCES environment variable, 143
SPARK_EXECUTOR_MEMORY environment variable, 143
SPARK_HOME directory, 36–37
SPARK_HOME environment variable, 33, 142
SPARK_LOCAL_IP environment variable, 144
SPARK_MASTER_IP environment variable, 144
SPARK_MASTER_OPTS environment variable, 144
SPARK_MASTER_PORT environment variable, 144
SPARK_MASTER_WEBUI_PORT environment variable, 144
SPARK_PUBLIC_DNS environment variable, 144
SPARK_WORKER_CORES environment variable, 144
SPARK_WORKER_DIR environment variable, 144
SPARK_WORKER_INSTANCES environment variable, 144
SPARK_WORKER_MEMORY environment variable, 144
SPARK_WORKER_OPTS environment variable, 144
SPARK_WORKER_PORT environment variable, 144
SPARK_WORKER_WEBUI_PORT environment variable, 144
SPARK_YARN_APP_NAME environment variable, 143
SPARK_YARN_DIST_ARCHIVES environment variable, 143
SPARK_YARN_DIST_FILES environment variable, 143
SPARK_YARN_QUEUE environment variable, 143
spark.broadcast.blockSize option, 114
spark.broadcast.compress option, 114
spark.broadcast.factory option, 114
spark.broadcast.port option, 114
SparkConf, 47
SparkContext, 47, 211
spark.default.parallelism property, 121
spark-defaults.conf file, 38
spark.driver.extraClassPath property, 145–147
spark.driver.extraJavaOptions property, 145–147
spark.driver.memory property, 145–147
spark.dynamicAllocation.enabled property, 145–147
spark-env.sh file, 38
spark.executor.cores property, 145–147
spark.executor.extraClassPath property, 145–147
spark.executor.extraJavaOptions property, 145–147
spark.executor.memory property, 145–147
spark-hbase-connector, 201
spark.master property, 145–147
%spark.pyspark interpreter, 279
SparkR, 243, 250
  accessing, 250–251
  data frames, creating, 251–253
  data mining, 254–255
  documentation, 250
  linear regression, 255–256
  predictive analytics, 253–254
  predictive modeling, 254–255
  RStudio and, 257–258
SPARKR_DRIVER_R environment variable, 142
sparkR.session() function, 253
SparkSession, 46–48, 167–168, 211, 250
spark-shell command, 30
spark.shuffle.service.enabled property, 145–147
%spark.sql interpreter, 279
spark-sql shell, 191
spark-streaming-kafka-assembly.jar file, 232
spark-submit command, 16, 30, 55–56, 192
SparseVector object, 271
sparsity, HBase support for, 199
splittable compression formats, 62
splitting datasets, 263–264
SQL (Structured Query Language). *See* Spark SQL
sql() function, 170, 253
SQLContext, 47
Sqoop, 8
SQS (Simple Queue Service), 228
square brackets ([ ]), 19
stages
  defined, 48
  dependencies, 48
Standalone deployment mode (Spark), 28–29

Standalone scheduler, 53
daemon environment variables, 144
multi-node standalone clusters, 37–39
Standalone deployment mode, 28–29
start() method, 211
state operations (Spark Streaming), 219–221
statements
CREATE TABLE, 165
def, 23
DROP TABLE, 165
UPDATE, 163
statistical functions, 184, 249
statistics, partition, 166
stats() transformation, 108
stdev() transformation, 107
stop() method, 211
stop-word-list.csv file, 119
storage, 173
columnar, 166
getStorageLevel() function, 130
storage levels
choosing, 131
table of, 128–129
storage-level flags, 129–130
StorageClass constructor, 129
storage-level flags, 129–130
stream processing. *See* **Spark Streaming**
StreamingContext, 47, 211, 215
streaming-wordcount folder, 219
STRING datatype, 165
string datatypes
Hive, 165
Spark primitive type, 176
string functions, 184
StringType, 176
STRUCT datatype, 165
struct datatypes
Hive, 165
Spark primitive type, 176
StructField objects, 178
StructType, 176, 178
Structured Query Language. *See* **Spark SQL**
Structured Streaming, 223–224
data sinks, 225
console sinks, 226
file sinks, 225–226
memory sinks, 226
data sources, 224
file sources, 224–225
socket sources, 225
operations, 227–228
structures, data. *See* **data structures**
submitting Spark jobs
interactive, 15
in Local mode, 28
to Mesos cluster, 30
non-interactive, 16
to standalone cluster, 29
to YARN cluster, 30
subtract() function, 104–105, 186
subtractByKey() transformation, 105
sum() transformation, 107
summary() function, 255–256
supervised learning, 254

T

table() function, 170–171
tables
Cassandra, 202
DynamoDB, 204
HBase, 198
Hive
creating data frames from, 170–171, 253
writing DataFrame output to, 188
tablets, 199
take() action, 82, 152
takeSample() function, 140–141, 152
tasks
defined, 48
optimizing execution of, 156–157
Tensor Processing Units (TPUs), 41
TensorFlow, 41
Term Frequency-Inverse Document Frequency (TF-IDF), 92, 261
testing
multi-node standalone clusters, 38
Spark installation
on Linux or Mac OS X, 33
on Windows, 35
text files
creating RDDs from, 63
textFile() method, 63–64
wholeTextFiles() method, 64–66
saving RDDs as, 217

text() method, 173
textFile() method, 17, 63–64
Tez, 173
TF-IDF (Term Frequency-Inverse Document Frequency), 92, 261
Thrift JDBC/ODBC server
  overview of, 192
  sample exercise, 194–195
TIBCO EMS (Enterprise Message Service), 228
TIMESTAMP datatype, 165
timestamp types
  Hive, 165
  Spark primitive, 176
TimestampType, 176
TINYINT datatype, 165
toDebugString() function, 128
Toffler, Alvin, 111
top() action, 82–83
Torvalds, Linus, 59
TPUs (Tensor Processing Units), 41
train() method, 266
trainClassifier() function, 265
transform() method, 274
transformations, 48, 77
  cartesian(), 99–100
  coarse-grained, 72
  cogroup(), 98–99
  defined, 59
  distinct(), 79–80
  DStreams, 213–214
  example of, 72
  filter(), 79
  fine-grained, 72
  flatMap(), 78
  flatMapValues(), 87–89
  foldByKey(), 91
  fullOuterJoin(), 98
  groupBy(), 80
  groupByKey(), 89
  intersection(), 104
  join(), 96–97
  keyBy(), 86–87
  keys(), 86
  leftOuterJoin(), 97
  map(), 77–78
  mapValues(), 87
  max(), 106
  mean(), 106
  min(), 105–106
  reduceByKey(), 90
  rightOuterJoin(), 97
  sortBy(), 81
  sortByKey(), 91–92
  stats(), 108
  stdev(), 107
  subtract(), 104–105
  subtractByKey(), 105
  sum(), 107
  union(), 104
  values(), 86
  variance(), 107
Transformer objects, 274
treeAggregate() function, 150
treeReduce() function, 150
tuple() function, 19
tuples, 18–19

U

udf() method, 184–185
UDFs (user-defined functions), 184–185
union() transformation, 104
UnionRDD, 77
unpersist() method, 114–116, 132–134, 187
unsupervised learning, 254
update output mode, 227
UPDATE statement, 163
updateStateByKey() method, 220–221
URI structures, 63
urllib2 Python module, 119
user-defined functions (UDFs), 184–185

V

value() method, 113, 117
values() method, 20, 86
var structure, 245–246
variables
  environment, 141–142
    defaults, 147
    HADOOP_CONF_DIR, 142
    HADOOP_HOME, 35, 142
    HIVE_CONF_DIR, 143
    JAVA_HOME, 142
    PYSPARK_DRIVER_PYTHON, 142
    PYSPARK_PYTHON, 142
    SPARK_CLASSPATH, 144
    SPARK_DAEMON_MEMORY, 144

SPARK_DRIVER_MEMORY, 143
SPARK_EXECUTOR_CORES, 143
SPARK_EXECUTOR_INSTANCES, 143
SPARK_EXECUTOR_MEMORY, 143
SPARK_HOME, 33, 142
SPARK_LOCAL_IP, 144
SPARK_MASTER_IP, 144
SPARK_MASTER_OPTS, 144
SPARK_MASTER_PORT, 144
SPARK_MASTER_WEBUI_PORT, 144
SPARK_PUBLIC_DNS, 144
SPARK_WORKER_CORES, 144
SPARK_WORKER_DIR, 144
SPARK_WORKER_INSTANCES, 144
SPARK_WORKER_MEMORY, 144
SPARK_WORKER_OPTS, 144
SPARK_WORKER_PORT, 144
SPARK_WORKER_WEBUI_PORT, 144
SPARK_YARN_APP_NAME, 143
SPARK_YARN_DIST_ARCHIVES, 143
SPARK_YARN_DIST_FILES, 143
SPARK_YARN_QUEUE, 143
SPARKR_DRIVER_R, 142
YARN_CONF_DIR, 142
R, 252
shared, 111
accumulators, 116–119
broadcast variables, 112–116
sample exercise, 119–120
variance() transformation, 107
vectors, 245–246
Venn diagrams, 103
Virtual Private Cloud (VPC), 39
VPC (Virtual Private Cloud), 39

W

Waikato Environment for Knowledge Analysis (WEKA), 263
WALs (write-ahead logs), 229
weather dataset, 262–263
WEKA (Waikato Environment for Knowledge Analysis), 263
wget command, 33
wholeTextFiles() method, 64–66, 76
wide operations, 75
window() method, 222–223
windowing functions, 184
windows, sliding window operations, 221
reduceByKeyAndWindow() method, 223
window() method, 222–223
Windows, Spark installation on, 34–36
winutils.exe, 34
WordCount exercise, 92–95
Workers, 38, 49–51
write operations (HDFS), 9–10
write-ahead logs (WALs), 229
write.csv() method, 189–190
write.parquet() method, 190–191
writeStream() method, 227
writing DataFrame output
to files, 188–191
to Hive tables, 188

X-Y

Yahoo!, 6
YARN (Yet Another Resource Negotiator), 7–8
application scheduling with, 10–13
ApplicationMaster, 11–12
applications running on, 53–57
ApplicationMaster, 53
deployment modes, 53–57
ResourceManager, 53
environment variables, 143
NodeManagers, 10–12
as resource scheduler for Spark, 17
ResourceManager, 10–12
Spark jobs, submitting, 30
Spark on, 28–29
yarn/ directory, 37
YARN_CONF_DIR environment variable, 142
Yet Another Resource Negotiator. *See* **YARN (Yet Another Resource Negotiator)**

Z

Zaharia, Matei, 13
Zeppelin, 40–41
interpreters, 279
notebooks, 278–279
zero() function, 118
ZeroMQ (ØMQ), 228
ZIP format, 62
ZooKeeper, 230, 234–237