GETTING STARTED WITH

APACHE SPARK

From Inception to Production

James A. Scott

Getting Started with Apache Spark

Inception to Production

James A. Scott

Getting Started with Apache Spark

by James A. Scott

Printed in the United States of America

Published by MapR Technologies, Inc., 350 Holger Way, San Jose, CA 95134

September 2015: First Edition

Revision History for the First Edition:

2015-09-01: First release

Table of Contents

1 What is Apache Spark

A new name has entered many of the conversations around big data recently. Some see the popular newcomer Apache Spark™ as a more accessible and more powerful replacement for Hadoop, big data's original technology of choice. Others recognize Spark as a powerful complement to Hadoop and other more established technologies, with its own set of strengths, quirks and limitations.

Spark, like other big data tools, is powerful, capable, and well-suited to tackling a range of data challenges. Spark, like other big data technologies, is not necessarily the best choice for every data processing task.

In this report, we introduce Spark and explore some of the areas in which its particular set of capabilities show the most promise. We discuss the relationship to Hadoop and other key technologies, and provide some helpful pointers so that you can hit the ground running and confidently try Spark for yourself.

What is Spark?

Spark began life in 2009 as a project within the AMPLab at the University of California, Berkeley. More specifically, it was born out of the necessity to prove out the concept of Mesos, which was also created in the AMPLab. Spark was first discussed in the Mesos white paper titled *Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center*, written most notably by Benjamin Hindman and Matei Zaharia.

From the beginning, Spark was optimized to run in memory, helping process data far more quickly than alternative approaches like Hadoop's MapReduce, which tends to write data to and from computer hard drives between each stage of processing. Its proponents claim that Spark running in memory can be 100 times faster than Hadoop MapReduce, but also 10 times faster when processing disk-based data in a similar way to Hadoop MapReduce itself. This comparison is not entirely fair, not least because raw speed tends to be more impor-

tant to Spark's typical use cases than it is to batch processing, at which MapReduce-like solutions still excel.

Spark became an incubated project of the Apache Software Foundation in 2013, and early in 2014, Apache Spark was promoted to become one of the Foundation's top-level projects. Spark is currently one of the most active projects managed by the Foundation, and the community that has grown up around the project includes both prolific individual contributors and well-funded corporate backers such as Databricks, IBM and China's Huawei.

Spark is a general-purpose data processing engine, suitable for use in a wide range of circumstances. Interactive queries across large data sets, processing of streaming data from sensors or financial systems, and machine learning tasks tend to be most frequently associated with Spark. Developers can also use it to support other data processing tasks, benefiting from Spark's extensive set of developer libraries and APIs, and its comprehensive support for languages such as Java, Python, R and Scala. Spark is often used alongside Hadoop's data storage module, HDFS, but can also integrate equally well with other popular data storage subsystems such as HBase, Cassandra, MapR-DB, MongoDB and Amazon's S3.

There are many reasons to choose Spark, but three are key:

- **Simplicity**: Spark's capabilities are accessible via a set of rich APIs, all designed specifically for interacting quickly and easily with data at scale. These APIs are well documented, and structured in a way that makes it straightforward for data scientists and application developers to quickly put Spark to work;
- **Speed**: Spark is designed for speed, operating both in memory and on disk. In 2014, Spark was used to win the **Daytona Gray Sort benchmarking challenge**, processing 100 terabytes of data stored on solid-state drives in just 23 minutes. The previous winner used Hadoop and a different cluster configuration, but it took 72 minutes. This win was the result of processing a static data set. Spark's performance can be even greater when supporting interactive queries of data stored in memory, with claims that Spark can be 100 times faster than Hadoop's MapReduce in these situations;
- **Support**: Spark supports a range of programming languages, including Java, Python, R, and Scala. Although often closely associated with Hadoop's underlying storage system, HDFS, Spark includes native support for tight integration with a number of leading storage solutions in the Hadoop ecosystem and beyond. Additionally, the Apache Spark community is large, active, and international. A growing set of commercial providers

including Databricks, IBM, and all of the main Hadoop vendors deliver comprehensive support for Spark-based solutions.

Who Uses Spark?

A wide range of technology vendors have been quick to support Spark, recognizing the opportunity to extend their existing big data products into areas such as interactive querying and machine learning, where Spark delivers real value. Well-known companies such as IBM and Huawei have invested significant sums in the technology, and a growing number of startups are building businesses that depend in whole or in part upon Spark. In 2013, for example, the Berkeley team responsible for creating Spark founded Databricks, which provides a hosted end-to-end data platform powered by Spark.

The company is well-funded, having received $47 million across two rounds of investment in 2013 and 2014, and Databricks employees continue to play a prominent role in improving and extending the open source code of the Apache Spark project.

The major Hadoop vendors, including MapR, Cloudera and Hortonworks, have all moved to support Spark alongside their existing products, and each is working to add value for their customers.

Elsewhere, IBM, Huawei and others have all made significant investments in Apache Spark, integrating it into their own products and contributing enhancements and extensions back to the Apache project.

Web-based companies like Chinese search engine Baidu, e-commerce operation Alibaba Taobao, and social networking company Tencent all run Spark-based operations at scale, with Tencent's 800 million active users reportedly generating over 700 TB of data per day for processing on a cluster of more than 8,000 compute nodes.

In addition to those web-based giants, pharmaceutical company Novartis depends upon Spark to reduce the time required to get modeling data into the hands of researchers, while ensuring that ethical and contractual safeguards are maintained.

What is Spark Used For?

Spark is a general-purpose data processing engine, an API-powered toolkit which data scientists and application developers incorporate into their applications to rapidly query, analyze and transform data at scale. Spark's flexibility makes it well-suited to tackling a range of use cases, and it is capable of handling several petabytes of data at a time, distributed across a cluster of thousands of cooperating physical or virtual servers. Typical use cases include:

- **Stream processing**: From log files to sensor data, application developers increasingly have to cope with "streams" of data. This data arrives in a steady stream, often from multiple sources simultaneously. While it is certainly feasible to allow these data streams to be stored on disk and analyzed retrospectively, it can sometimes be sensible or important to process and act upon the data as it arrives. Streams of data related to financial transactions, for example, can be processed in real time to identify--and refuse--potentially fraudulent transactions.
- **Machine learning**: As data volumes grow, machine learning approaches become more feasible and increasingly accurate. Software can be trained to identify and act upon triggers within well-understood data sets before applying the same solutions to new and unknown data. Spark's ability to store data in memory and rapidly run repeated queries makes it well-suited to training machine learning algorithms. Running broadly similar queries again and again, at scale, significantly reduces the time required to iterate through a set of possible solutions in order to find the most efficient algorithms.
- **Interactive analytics**: Rather than running pre-defined queries to create static dashboards of sales or production line productivity or stock prices, business analysts and data scientists increasingly want to explore their data by asking a question, viewing the result, and then either altering the initial question slightly or drilling deeper into results. This interactive query process requires systems such as Spark that are able to respond and adapt quickly.
- **Data integration**: Data produced by different systems across a business is rarely clean or consistent enough to simply and easily be combined for reporting or analysis. Extract, transform, and load (ETL) processes are often used to pull data from different systems, clean and standardize it, and then load it into a separate system for analysis. Spark (and Hadoop) are increasingly being used to reduce the cost and time required for this ETL process.

How to Install Apache Spark 2

Although cluster-based installations of Spark can become large and relatively complex by integrating with Mesos, Hadoop, Cassandra, or other systems, it is straightforward to download Spark and configure it in standalone mode on a laptop or server for learning and exploration. This low barrier to entry makes it relatively easy for individual developers and data scientists to get started with Spark, and for businesses to launch pilot projects that do not require complex re-tooling or interference with production systems.

Apache Spark is open source software, and can be freely **downloaded** from the Apache Software Foundation. Spark requires at least version 6 of Java, and at least version 3.0.4 of Maven. Other dependencies, such as Scala and Zinc, are automatically installed and configured as part of the installation process.

Build options, including optional links to data storage systems such as Hadoop's HDFS or Hive, are discussed in more detail in Spark's online documentation.

A **Quick Start** guide, optimized for developers familiar with either Python or Scala, is an accessible introduction to working with Spark.

One of the simplest ways to get up and running with Spark is to use the **MapR Sandbox** which includes Spark. MapR provides a **tutorial** linked to their simplified deployment of Hadoop.

A Very Simple Spark Installation

Follow these simple steps to download Java, Spark, and Hadoop and get them running on a laptop (in this case, one running Mac OS X). If you do not currently have the **Java JDK** (version 7 or higher) installed, download it and follow the steps to install it for your operating system.

Visit the **Spark downloads page**, select a pre-built package, and download Spark. Double-click the archive file to expand its contents ready for use.

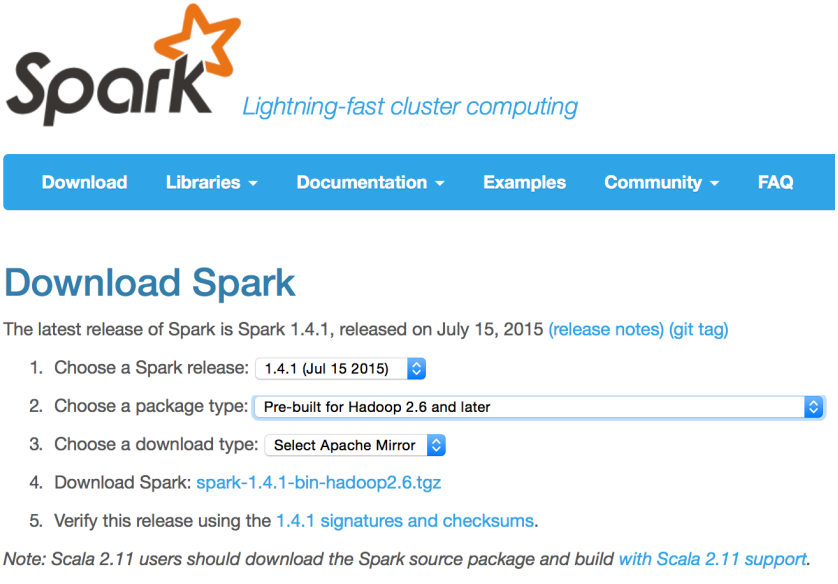

FIGURE 2-1
Apache Spark download page, with a pre-built package

Testing Spark

Open a text console, and navigate to the newly created directory. Start Spark's interactive shell:

```
./bin/spark-shell
```

A series of messages will scroll past as Spark and Hadoop are configured. Once the scrolling stops, you will see a simple prompt.

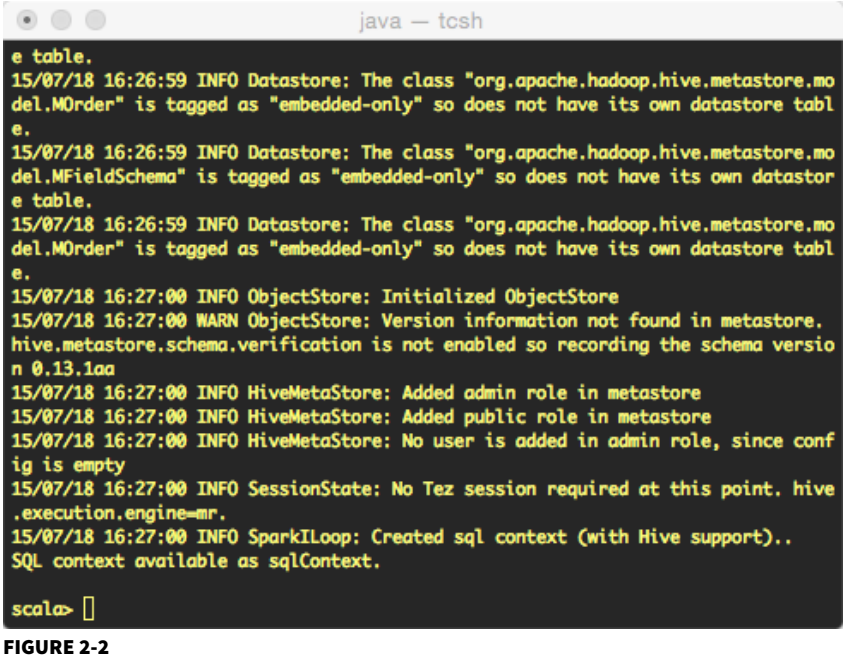

FIGURE 2-2

Terminal window after Spark starts running

At this prompt, let's create some data; a simple sequence of numbers from 1 to 50,000.

```
val data = 1 to 50000
```

Now, let's place these 50,000 numbers into a Resilient Distributed Dataset (RDD) which we'll call sparkSample. It is this RDD upon which Spark can perform analysis.

```
val sparkSample = sc.parallelize(data)
```

Now we can filter the data in the RDD to find any values of less than 10.

```
sparkSample.filter(_ < 10).collect()
```

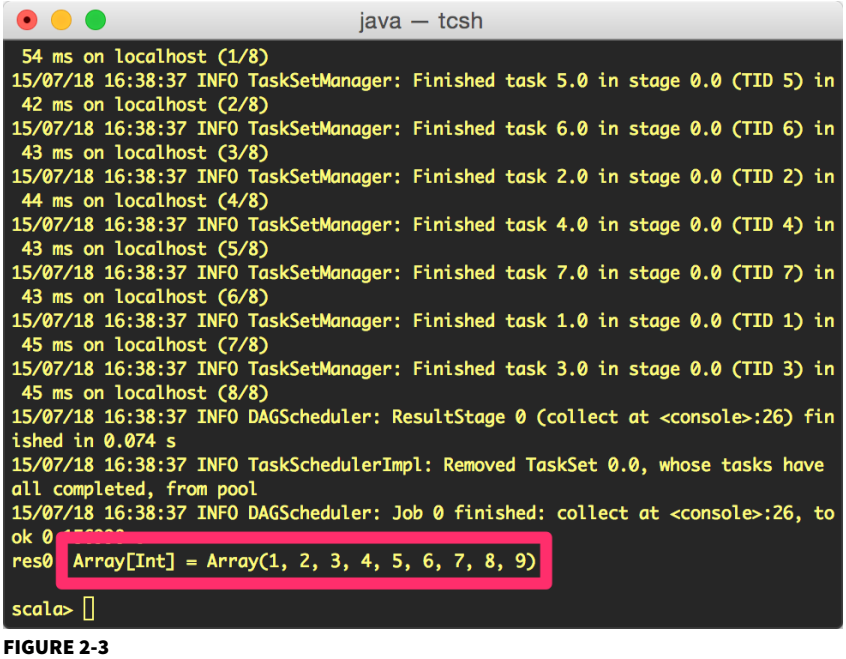

FIGURE 2-3

Values less than 10, from a set of 50,000 numbers

Spark should report the result, with an array containing any values less than 10. Richer and more complex examples are available in resources mentioned elsewhere in this guide.

Spark has a very low entry barrier to get started, which eases the burden of learning a new toolset. Barrier to entry should always be a consideration for any new technology a company evaluates for enterprise use.

Apache Spark Architectural Overview 3

Spark is a top-level project of the Apache Software Foundation, designed to be used with a range of programming languages and on a variety of architectures. Spark's speed, simplicity, and broad support for existing development environments and storage systems make it increasingly popular with a wide range of developers, and relatively accessible to those learning to work with it for the first time. The project supporting Spark's ongoing development is one of Apache's largest and most vibrant, with over 500 contributors from more than 200 organizations responsible for code in the current software release.

Development Language Support

Comprehensive support for the development languages with which developers are already familiar is important so that Spark can be learned relatively easily, and incorporated into existing applications as straightforwardly as possible. Programming languages supported by Spark include:

- **Java**
- **Python**
- **Scala**
- **SQL**
- **R**

Languages like Python are often regarded as poorly performing languages, especially in relation to alternatives such as Java. Although this concern is justified in some development environments, it is less significant in the distributed cluster model in which Spark will typically be deployed. Any slight loss of performance introduced by the use of Python can be compensated for elsewhere in the design and operation of the cluster. Familiarity with your chosen lan-

guage is likely to be far more important than the raw speed of code prepared in that language.

Extensive examples and tutorials exist for Spark in a number of places, including the **Apache Spark project website** itself. These tutorials normally include code snippets in Java, Python and Scala.

The Structured Query Language, SQL, is widely used in relational databases, and simple SQL queries are normally well-understood by developers, data scientists and others who are familiar with asking questions of any data storage system. The Apache Spark module--Spark SQL--offers native support for SQL and simplifies the process of querying data stored in Spark's own Resilient Distributed Dataset model, alongside data from external sources such as relational databases and data warehouses.

Support for the data science package, R, is more recent. The SparkR package first appeared in release 1.4 of Apache Spark (in June 2015), but given the popularity of R among data scientists and statisticians, it is likely to prove an important addition to Spark's set of supported languages.

Deployment Options

As noted in the previous chapter, Spark is easy to download and install on a laptop or virtual machine. Spark was built to be able to run in a couple different ways: standalone, or part of a cluster.

But for production workloads that are operating at scale, a single laptop or virtual machine is not likely to be sufficient. In these circumstances, Spark will normally run on an existing big data cluster. These clusters are often also used for Hadoop jobs, and Hadoop's YARN resource manager will generally be used to manage that Hadoop cluster (including Spark). **Running Spark on YARN**, from the Apache Spark project, provides more configuration details.

For those who prefer alternative resource managers, Spark can also run just as easily on clusters controlled by Apache Mesos. **Running Spark on Mesos**, from the Apache Spark project, provides more configuration details.

A series of scripts bundled with current releases of Spark simplify the process of launching Spark on Amazon Web Services' Elastic Compute Cloud (EC2). **Running Spark on EC2**, from the Apache Spark project, provides more configuration details.

Storage Options

Although often linked with the Hadoop Distributed File System (HDFS), Spark can integrate with a range of commercial or open source third-party data storage systems, including:

- MapR (file system and database)
- Google Cloud
- Amazon S3
- Apache Cassandra
- Apache Hadoop (HDFS)
- Apache HBase
- Apache Hive
- Berkeley's Tachyon project

Developers are most likely to choose the data storage system they are already using elsewhere in their workflow.

The Spark Stack

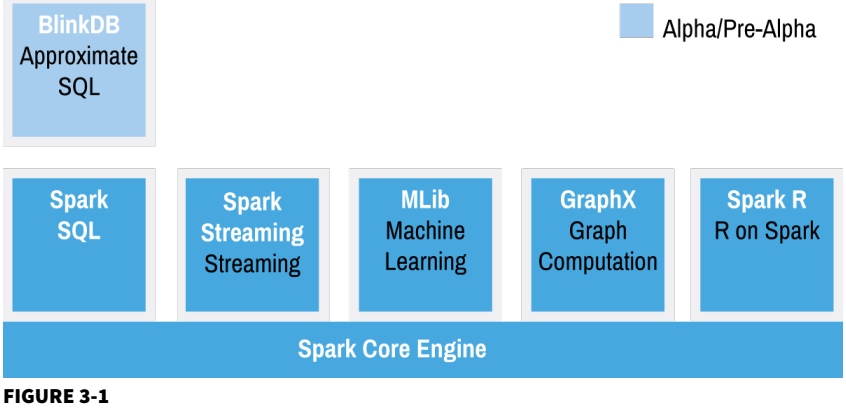

FIGURE 3-1
Spark Stack Diagram

The Spark project stack currently is comprised of Spark Core and four libraries that are optimized to address the requirements of four different use cases. Individual applications will typically require Spark Core and at least one of these libraries. Spark's flexibility and power become most apparent in applications that require the combination of two or more of these libraries on top of Spark Core.

- **Spark Core**: This is the heart of Spark, and is responsible for management functions such as task scheduling. Spark Core implements and de-

pends upon a programming abstraction known as Resilient Distributed Datasets (RDDs), which are discussed in more detail below.

- **Spark SQL**: This is Spark's module for working with structured data, and it is designed to support workloads that combine familiar SQL database queries with more complicated, algorithm-based analytics. Spark SQL supports the open source Hive project, and its SQL-like HiveQL query syntax. Spark SQL also supports JDBC and ODBC connections, enabling a degree of integration with existing databases, data warehouses and business intelligence tools. JDBC connectors can also be used to integrate with Apache Drill, opening up access to an even broader range of data sources.
- **Spark Streaming**: This module supports scalable and fault-tolerant processing of streaming data, and can integrate with established sources of data streams like Flume (optimized for data logs) and Kafka (optimized for distributed messaging). Spark Streaming's design, and its use of Spark's RDD abstraction, are meant to ensure that applications written for streaming data can be repurposed to analyze batches of historical data with little modification.
- **MLlib**: This is Spark's scalable machine learning library, which implements a set of commonly used machine learning and statistical algorithms. These include correlations and hypothesis testing, classification and regression, clustering, and principal component analysis.
- **GraphX**: This module began life as a separate UC Berkeley research project, which was eventually donated to the Apache Spark project. GraphX supports analysis of and computation over graphs of data, and supports a version of graph processing's Pregel API. GraphX includes a number of widely understood graph algorithms, including PageRank.
- **Spark R**: This module was added to the 1.4.x release of Apache Spark, providing data scientists and statisticians using R with a lightweight mechanism for calling upon Spark's capabilities.

Resilient Distributed Datasets (RDDs)

The Resilient Distributed Dataset is a concept at the heart of Spark. It is designed to support in-memory data storage, distributed across a cluster in a manner that is demonstrably both fault-tolerant and efficient. Fault-tolerance is achieved, in part, by tracking the lineage of transformations applied to coarse-grained sets of data. Efficiency is achieved through parallelization of processing

across multiple nodes in the cluster, and minimization of data replication between those nodes. Once data is loaded into an RDD, two basic types of operation can be carried out:

- **Transformations**, which create a new RDD by changing the original through processes such as mapping, filtering, and more;
- **Actions**, such as counts, which measure but do not change the original data.

The original RDD remains unchanged throughout. The chain of transformations from RDD1 to RDDn are logged, and can be repeated in the event of data loss or the failure of a cluster node.

Transformations are said to be lazily evaluated, meaning that they are not executed until a subsequent action has a need for the result. This will normally improve performance, as it can avoid the need to process data unnecessarily. It can also, in certain circumstances, introduce processing bottlenecks that cause applications to stall while waiting for a processing action to conclude.

Where possible, these RDDs remain in memory, greatly increasing the performance of the cluster, particularly in use cases with a requirement for iterative queries or processes.

API Overview

Spark's capabilities can all be accessed and controlled using a rich API. This supports Spark's four principal development environments: (**Scala**, **Java**, **Python**, **R**), and extensive documentation is provided regarding the API's instantiation in each of these languages. The **Spark Programming Guide** provides further detail, with comprehensive code snippets in Scala, Java and Python. The Spark API was optimized for manipulating data, with a design that reduced common data science tasks from hundreds or thousands of lines of code to only a few.

An additional DataFrames API was added to Spark in 2015. DataFrames **offer**

- Ability to scale from kilobytes of data on a single laptop to petabytes on a large cluster
- Support for a wide array of data formats and storage systems
- State-of-the-art optimization and code generation through the Spark SQL Catalyst optimizer
- Seamless integration with all big data tooling and infrastructure via Spark
- APIs for Python, Java, Scala, and R

For those familiar with a DataFrames API in other languages like R or pandas in Python, this API will make them feel right at home. For those not familiar with the API, but already familiar with Spark, this extended API will ease application development, while helping to improve performance via the optimizations and code generation.

The Power of Data Pipelines

Much of Spark's power lies in its ability to combine very different techniques and processes together into a single, coherent, whole. Outside Spark, the discrete tasks of selecting data, transforming that data in various ways, and analyzing the transformed results might easily require a series of separate processing frameworks such as Apache Oozie. Spark, on the other hand, offers the ability to combine these together, crossing boundaries between batch, streaming and interactive workflows in ways that make the user more productive.

Spark jobs perform multiple operations consecutively, in memory and only spilling to disk when required by memory limitations. Spark simplifies the management of these disparate processes, offering an integrated whole--a data pipeline that is easier to configure, easier to run, and easier to maintain. In use cases such as ETL, these pipelines can become extremely rich and complex, combining large numbers of inputs and a wide range of processing steps into a unified whole that consistently delivers the desired result.

Benefits of Hadoop and Spark 4

Spark is a general-purpose data processing engine, suitable for use in a wide range of circumstances. In its current form, however, Spark is not designed to deal with the data management and cluster administration tasks associated with running data processing and analysis workloads at scale.

Rather than investing effort in building these capabilities into Spark, the project currently leverages the strengths of other open source projects, relying upon them for everything from cluster management and data persistence to disaster recovery and compliance.

Projects like Apache Mesos offer a powerful and growing set of capabilities around distributed cluster management. However, most Spark deployments today still tend to use Apache Hadoop and its associated projects to fulfill these requirements.

Hadoop vs. Spark – An Answer to the Wrong Question

Spark is not, despite the hype, a replacement for Hadoop. Nor is MapReduce dead.

Spark can run on top of Hadoop, benefiting from Hadoop's cluster manager (YARN) and underlying storage (HDFS, HBase, etc.). Spark can also run completely separately from Hadoop, integrating with alternative cluster managers like Mesos and alternative storage platforms like Cassandra and Amazon S3.

Much of the confusion around Spark's relationship to Hadoop dates back to the early years of Spark's development. At that time, Hadoop relied upon MapReduce for the bulk of its data processing. Hadoop MapReduce also managed scheduling and task allocation processes within the cluster; even workloads that were not best suited to batch processing were passed through Hadoop's MapReduce engine, adding complexity and reducing performance.

MapReduce is really a programming model. In Hadoop MapReduce, multiple MapReduce jobs would be strung together to create a data pipeline. In between

every stage of that pipeline, the MapReduce code would read data from the disk, and when completed, would write the data back to the disk. This process was inefficient because it had to read all the data from disk at the beginning of each stage of the process. This is where Spark comes in to play. Taking the same MapReduce programming model, Spark was able to get an immediate 10x increase in performance, because it didn't have to store the data back to the disk, and all activities stayed in memory. Spark offers a far faster way to process data than passing it through unnecessary Hadoop MapReduce processes.

Hadoop has since moved on with the development of the YARN cluster manager, thus freeing the project from its early dependence upon Hadoop MapReduce. Hadoop MapReduce is still available within Hadoop for running static batch processes for which MapReduce is appropriate. Other data processing tasks can be assigned to different processing engines (including Spark), with YARN handling the management and allocation of cluster resources.

Spark is a viable alternative to Hadoop MapReduce in a range of circumstances. Spark is not a replacement for Hadoop, but is instead a great companion to a modern Hadoop cluster deployment.

What Hadoop Gives Spark

Apache Spark is often deployed in conjunction with a Hadoop cluster, and Spark is able to benefit from a number of capabilities as a result. On its own, Spark is a powerful tool for processing large volumes of data. But, on its own, Spark is not yet well-suited to production workloads in the enterprise. Integration with Hadoop gives Spark many of the capabilities that broad adoption and use in production environments will require, including:

- **YARN resource manager**, which takes responsibility for scheduling tasks across available nodes in the cluster;
- **Distributed File System**, which stores data when the cluster runs out of free memory, and which persistently stores historical data when Spark is not running;
- **Disaster Recovery capabilities**, inherent to Hadoop, which enable recovery of data when individual nodes fail. These capabilities include basic (but reliable) data mirroring across the cluster and richer snapshot and mirroring capabilities such as those offered by the MapR Data Platform;
- **Data Security**, which becomes increasingly important as Spark tackles production workloads in regulated industries such as healthcare and financial services. Projects like Apache Knox and Apache Ranger offer data security capabilities that augment Hadoop. Each of the big three vendors

have alternative approaches for security implementations that complement Spark. Hadoop's core code, too, is increasingly recognizing the need to expose advanced security capabilities that Spark is able to exploit;

- **A distributed data platform**, benefiting from all of the preceding points, meaning that Spark jobs can be deployed on available resources anywhere in a distributed cluster, without the need to manually allocate and track those individual jobs.

What Spark Gives Hadoop

Hadoop has come a long way since its early versions which were essentially concerned with facilitating the batch processing of MapReduce jobs on large volumes of data stored in HDFS. Particularly since the introduction of the YARN resource manager, Hadoop is now better able to manage a wide range of data processing tasks, from batch processing to streaming data and graph analysis.

Spark is able to contribute, via YARN, to Hadoop-based jobs. In particular, Spark's machine learning module delivers capabilities not easily exploited in Hadoop without the use of Spark. Spark's original design goal, to enable rapid in-memory processing of sizeable data volumes, also remains an important contribution to the capabilities of a Hadoop cluster.

In certain circumstances, Spark's SQL capabilities, streaming capabilities (otherwise available to Hadoop through Storm, for example), and graph processing capabilities (otherwise available to Hadoop through Neo4J or Giraph) may also prove to be of value in enterprise use cases.

Solving Business Problems with Spark 5

Now that you have learned how to get Spark up and running, it's time to put some of this practical knowledge to use. The use cases and code examples described in this chapter are reasonably short and to the point. They are intended to provide enough context on the problem being described so they can be leveraged for solving many more problems.

If these use cases are not complicated enough for your liking, don't fret, as there are more in-depth use cases provided at the end of the book. Those use cases are much more involved, and get into more details and capabilities of Spark.

The first use case walks through loading and querying tabular data. This example is a foundational construct of loading data in Spark. This will enable you to understand how Spark gets data from disk, as well as how to inspect the data and run queries of varying complexity.

The second use case here is about building user profiles from a music streaming service. User profiles are used across almost all industries. The concept of a customer 360 is based on a user profile. The premise behind a user profile is to build a dossier about a user. Whether or not the user is a known person or just a number in a database is usually a minor detail, but one that would fall into areas of privacy concern. User profiles are also at the heart of all major digital advertising campaigns. One of the most common Internet-based scenarios for leveraging user profiles is to understand how long a user stays on a particular website. All-in-all, building user profiles with Spark is child's play.

Processing Tabular Data with Spark SQL

The examples here will help you get started using Apache Spark DataFrames with Scala. The new Spark DataFrames API is designed to make big data processing on tabular data easier. A Spark DataFrame is a distributed collection of data organized into named columns that provides operations to *filter*, *group*, or

compute aggregates, and can be used with Spark SQL. DataFrames can be constructed from structured data files, existing RDDs, or external databases.

Sample Dataset

The dataset to be used is from eBay online auctions. The eBay online auction dataset contains the following fields:

- **auctionid** - unique identifier of an auction
- **bid** - the proxy bid placed by a bidder
- **bidtime** - the time (in days) that the bid was placed, from the start of the auction
- **bidder** - eBay username of the bidder
- **bidderrate** - eBay feedback rating of the bidder
- **openbid** - the opening bid set by the seller
- **price** - the closing price that the item sold for (equivalent to the second highest bid + an increment)

The table below shows the fields with some sample data:

auctionid	bid	bidtime	bidder	bid-der-rate	open-bid	price	item	day-sto-live
8213034705	95	2.927373	jake7870	0	95	117.5	xbox	3

Using Spark DataFrames, we will explore the eBay data with questions like:

- How many auctions were held?
- How many bids were made per item?
- What's the minimum, maximum, and average number of bids per item?
- Show the bids with price > 100

Loading Data into Spark DataFrames

First, we will import some packages and instantiate a sqlContext, which is the entry point for working with structured data (rows and columns) in Spark and allows the creation of DataFrame objects.

```
//  SQLContext entry point for working with structured data
val sqlContext = new org.apache.spark.sql.SQLContext(sc)

// this is used to implicitly convert an RDD to a DataFrame.
import sqlContext.implicits._

// Import Spark SQL data types and Row.
import org.apache.spark.sql._
```

Start by loading the data from the ebay.csv file into a Resilient Distributed Dataset (RDD). RDDs have **transformations** and **actions**; the *first()* **action** returns the first element in the RDD:

```
// load the data into a  new RDD
val ebayText = sc.textFile("ebay.csv")

// Return the first element in this RDD
ebayText.first()
```

Use a Scala *case class* to define the Auction schema corresponding to the ebay.csv file. Then a *map()* **transformation** is applied to each element of *ebayText* to create the ebay RDD of Auction objects.

```
//define the schema using a case class
case class Auction(auctionid: String, bid: Float, bidtime: Float,
  bidder: String, bidderrate: Integer, openbid: Float, price:
Float,
  item: String, daystolive: Integer)

// create an RDD of Auction objects
val ebay = ebayText.map(_.split(",")).map(p => Auction(p(0),
  p(1).toFloat,p(2).toFloat,p(3),p(4).toInt,p(5).toFloat,
  p(6).toFloat,p(7),p(8).toInt))
```

Calling *first()* **action** on the ebay RDD returns the first element in the RDD:

```
// Return the first element in this RDD
ebay.first()

// Return the number of elements in the RDD
ebay.count()
```

A DataFrame is a distributed collection of data organized into named columns. Spark SQL supports automatically converting an RDD containing case classes to a DataFrame with the method toDF():

```
// change ebay RDD of Auction objects to a DataFrame
val auction = ebay.toDF()
```

Exploring and Querying the eBay Auction Data

DataFrames provide a domain-specific language for structured data manipulation in Scala, Java, and Python; below are some examples with the auction DataFrame. The *show()* **action** displays the top 20 rows in a tabular form:

```
// Display the top 20 rows of DataFrame
auction.show()
```

DataFrame *printSchema()* displays the schema in a tree format:

```
// Return the schema of this DataFrame
auction.printSchema()
```

After a DataFrame is instantiated it can be queried. Here are some example using the Scala DataFrame API:

```
// How many auctions were held?
auction.select("auctionid").distinct.count

// How many bids per item?
auction.groupBy("auctionid", "item").count.show

// What's the min number of bids per item?
// what's the average? what's the max?
auction.groupBy("item", "auctionid").count
  .agg(min("count"), avg("count"),max("count")).show

// Get the auctions with closing price > 100
val highprice= auction.filter("price > 100")

// display dataframe in a tabular format
highprice.show()
```

A DataFrame can also be registered as a temporary table using a given name, which can then have SQL statements run against it using the methods provided by sqlContext. Here are some example queries using sqlContext:

```
// register the DataFrame as a temp table
auction.registerTempTable("auction")

// How many bids per auction?
val results = sqlContext.sql(
  "SELECT auctionid, item,  count(bid) FROM auction
    GROUP BY auctionid, item"
  )

// display dataframe in a tabular format
results.show()

val results = sqlContext.sql(
  "SELECT auctionid, MAX(price) FROM auction
    GROUP BY item,auctionid"
  )
results.show()
```

Summary

You have now learned how to load data into Spark DataFrames, and explore tabular data with Spark SQL. These code examples can be reused as the foundation to solve any type of business problem.

Computing User Profiles with Spark

This use case will bring together the core concepts of Spark and use a large dataset to build a simple real-time dashboard that provides insight into customer behaviors.

Spark is an enabling technology in a wide variety of use cases across many industries. Spark is a great candidate anytime results are needed fast and much of the computations can be done in memory. The language used here will be Python, because it does a nice job of reducing the amount of boilerplate code required to illustrate these examples.

Delivering Music

Music streaming is a rather pervasive technology which generates massive quantities of data. This type of service is much like people would use every day on a desktop or mobile device, whether as a subscriber or a free listener (perhaps even similar to a Pandora). This will be the foundation of the use case to be explored. Data from such a streaming service will be analyzed.

The basic layout consists of customers whom are logging into this service and listening to music tracks, and they have a variety of parameters:

- Demographic information (gender, location, etc.)
- Free / paid subscriber
- Listening history; tracks selected, when, and geolocation when they were selected

Python, PySpark and MLlib will be used to compute some basic statistics for a dashboard, enabling a high-level view of customer behaviors as well as a constantly updated view of the latest information.

Looking at the Data

This service has users whom are continuously connecting to the service and listening to tracks. Customers listening to music from this streaming service generate events, and over time they represent the highest level of detail about customers' behaviors.

The data will be loaded directly from a CSV file. There are a couple of steps to perform before it can be analyzed. The data will need to be transformed and loaded into a *PairRDD*. This is because the data consists of arrays of (key, value) tuples.

The customer events-individual tracks dataset (**tracks.csv**) consists of a collection of events, one per line, where each event is a client listening to a track. This size is approximately 1M lines and contains simulated listener events over several months. Because this represents things that are happening at a very low level, this data has the potential to grow very large.

Field Name	Event ID	Customer ID	Track ID	Datetime	Mobile	Listening Zip
Type	***Integer***	***Integer***	***Integer***	***String***	***Integer***	***Integer***
Example	9999767	2597	788	2014-12-01 09:54:09	0	11003

The event, customer and track IDs show that a customer listened to a specific track. The other fields show associated information, like whether the customer was listening on a mobile device, and a geolocation. This will serve as the input into the first Spark job.

The customer information dataset (**cust.csv**) consists of all statically known details about a user.

Field Name	Customer ID	Name	Gender	Address	Zip	Sign Date	Status	Level	Campaign	Linked with apps?
Type	***Integer***	***String***	***Integer***	***String***	***Integer***	***String***	***Integer***	***Integer***	***Integer***	***Integer***
Example	10	Joshua Threadgill	0	10084 Easy Gate Bend	66216	01/13/2013	0	1	1	1

The fields are defined as follows:

- **Customer ID**: a unique identifier for that customer
- **Name, gender, address, zip**: the customer's associated information
- **Sign date**: the date of addition to the service
- **Status**: indicates whether or not the account is active (0 = closed, 1 = active)
- **Level**: indicates what level of service -0, 1, 2 for Free, Silver and Gold, respectively
- **Campaign**: indicates the campaign under which the user joined, defined as the following (fictional) campaigns driven by our (also fictional) marketing team:
    - **NONE** no campaign

- **30DAYFREE** a '30 days free' trial offer
- **SUPERBOWL** a Super Bowl-related program
- **RETAILSTORE** an offer originating in brick-and-mortar retail stores
- **WEBOFFER** an offer for web-originated customers

Other datasets that would be available, but will not be used for this use case, would include:

- Advertisement click history
- Track details like title, album and artist

Customer Analysis

All the right information is in place and a lot of micro-level detail is available that describes what customers listen to and when. The quickest way to get this data to a dashboard is by leveraging Spark to create summary information for each customer as well as basic statistics about the entire user base. After the results are generated, they can be persisted to a file which can be easily used for visualization with BI tools such as Tableau, or other dashboarding frameworks like C3.js or D3.js.

Step one in getting started is to initialize a Spark context. Additional parameters could be passed to the *SparkConf* method to further configure the job, such as setting the master and the directory where the job executes.

```
from pyspark import SparkContext, SparkConf
from pyspark.mllib.stat import Statistics
import csv

conf = SparkConf().setAppName('ListenerSummarizer')
sc = SparkContext(conf=conf)
```

The next step will be to read the CSV records with the individual track events, and make a *PairRDD* out of all of the rows. To convert each line of data into an array, the *map()* function will be used, and then *reduceByKey()* is called to consolidate all of the arrays.

```
trackfile = sc.textFile('/tmp/data/tracks.csv')

def make_tracks_kv(str):
    l = str.split(",")
    return [l[1], [[int(l[2]), l[3], int(l[4]), l[5]]]]

    # make a k,v RDD out of the input data
    tbycust = trackfile.map(lambda line: make_tracks_kv(line))
      .reduceByKey(lambda a, b: a + b)
```

The individual track events are now stored in a *PairRDD*, with the customer ID as the key. A summary profile can now be computed for each user, which will include:

- Average number of tracks during each period of the day (time ranges are arbitrarily defined in the code)
- Total unique tracks, i.e., the set of unique track IDs
- Total mobile tracks, i.e., tracks played when the mobile flag was set

By passing a function to *mapValues*, a high-level profile can be computed from the components. The summary data is now readily available to compute basic statistics that can be used for display, using the *colStats* function from *pyspark.mllib.stat*.

```
def compute_stats_byuser(tracks):
    mcount = morn = aft = eve = night = 0
 tracklist = []
    for t in tracks:
        trackid, dtime, mobile, zip = t
        if trackid not in tracklist:
            tracklist.append(trackid)
        d, t = dtime.split(" ")
        hourofday = int(t.split(":")[0])
        mcount += mobile
        if (hourofday < 5):
            night += 1
        elif (hourofday < 12):
            morn += 1
        elif (hourofday < 17):
            aft += 1
        elif (hourofday < 22):
            eve += 1
        else:
            night += 1
        return [len(tracklist), morn, aft, eve, night, mcount]

# compute profile for each user
custdata = tbycust.mapValues(lambda a: compute_stats_byuser(a))

# compute aggregate stats for entire track history
aggdata = Statistics.colStats(custdata.map(lambda x: x[1]))
```

The last line provides meaningful statistics like the mean and variance for each of the fields in the per-user RDDs that were created in *custdata*.

Calling *collect()* on this RDD will persist the results back to a file. The results could be stored in a database such as MapR-DB, HBase or an RDBMS (using a

Python package like *happybase* or *dbset*). For the sake of simplicity for this example, using CSV is the optimal choice. There are two files to output:

- *live_table.csv* containing the latest calculations
- *agg_table.csv* containing the aggregated data about all customers computed with *Statistics.colStats*

```
for k, v in custdata.collect():
    unique, morn, aft, eve, night, mobile = v
    tot = morn + aft + eve + night

    # persist the data, in this case write to a file
with open('live_table.csv', 'wb') as csvfile:
        fwriter = csv.writer(csvfile, delimiter=' ',
               quotechar='|', quoting=csv.QUOTE_MINIMAL)
        fwriter.writerow(unique, morn, aft, eve, night, mobile)

        # do the same with the summary data
with open('agg_table.csv', 'wb') as csvfile:
        fwriter = csv.writer(csvfile, delimiter=' ',
               quotechar='|', quoting=csv.QUOTE_MINIMAL)
        fwriter.writerow(aggdata.mean()[0], aggdata.mean()[1],
            aggdata.mean()[2], aggdata.mean()[3], aggdata.mean()
[4],
            aggdata.mean()[5])
```

After the job completes, a summary is displayed of what was written to the CSV table and the averages for all users.

The Results

With just a few lines of code in Spark, a high-level customer behavior view was created, all computed using a dataset with millions of rows that stays current with the latest information. Nearly any toolset that can utilize a CSV file can now leverage this dataset for visualization.

This use case showcases how easy it is to work with Spark. Spark is a framework for ensuring that new capabilities can be delivered well into the future, as data volumes grow and become more complex.

Spark Streaming Framework and Processing Models

6

Although now considered a key element of Spark, streaming capabilities were only introduced to the project with its 0.7 release (February 2013), emerging from the alpha testing phase with the 0.9 release (February 2014). Rather than being integral to the design of Spark, stream processing is a capability that has been added alongside Spark Core and its original design goal of rapid in-memory data processing.

Other stream processing solutions exist, including projects like Apache Storm and Apache Flink. In each of these, stream processing is a key design goal, offering some advantages to developers whose sole requirement is the processing of data streams. These solutions, for example, typically process the data stream event-by-event, while Spark adopts a system of chopping the stream into chunks (or micro-batches) to maintain compatibility and interoperability with Spark Core and Spark's other modules.

The Details of Spark Streaming

Spark's real and sustained advantage over these alternatives is this tight integration between its stream and batch processing capabilities. Running in a production environment, Spark Streaming will normally rely upon capabilities from external projects like ZooKeeper and HDFS to deliver resilient scalability. In real-world application scenarios, where observation of historical trends often augments stream-based analysis of current events, this capability is of great value in streamlining the development process. For workloads in which streamed data must be combined with data from other sources, Spark remains a strong and credible option.

FIGURE 6-1
Data from a variety of sources to various storage systems

A streaming framework is only as good as its data sources. A strong messaging platform is the best way to ensure solid performance for any streaming system.

Spark Streaming supports the ingest of data from a wide range of data sources, including live streams from Apache Kafka, Apache Flume, Amazon Kinesis, Twitter, or sensors and other devices connected via TCP sockets. Data can also be streamed out of storage services such as HDFS and AWS S3. Data is processed by Spark Streaming, using a range of algorithms and high-level data processing functions like *map*, *reduce*, *join* and *window*. Processed data can then be passed to a range of external file systems, or used to populate live dashboards.

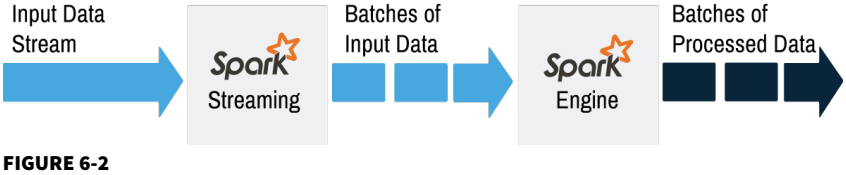

FIGURE 6-2
Incoming streams of data divided into batches

Logically, Spark Streaming represents a continuous stream of input data as a discretized stream, or DStream. Internally, Spark actually stores and processes this DStream as a sequence of RDDs. Each of these RDDs is a snapshot of all data ingested during a specified time period, which allows Spark's existing batch processing capabilities to operate on the data.

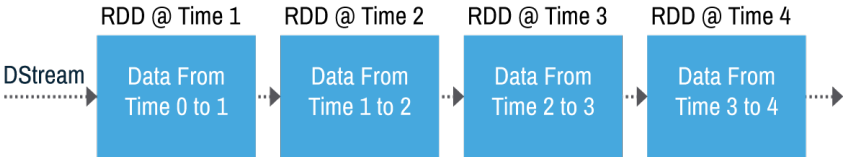

FIGURE 6-3

Input data stream divided into discrete chunks of data

The data processing capabilities in Spark Core and Spark's other modules are applied to each of the RDDs in a DStream in exactly the same manner as they would be applied to any other RDD: Spark modules other than Spark Streaming have no awareness that they are processing a data stream, and no need to know.

A basic RDD operation, *flatMap*, can be used to extract individual words from lines of text in an input source. When that input source is a data stream, *flatMap* simply works as it normally would, as shown below.

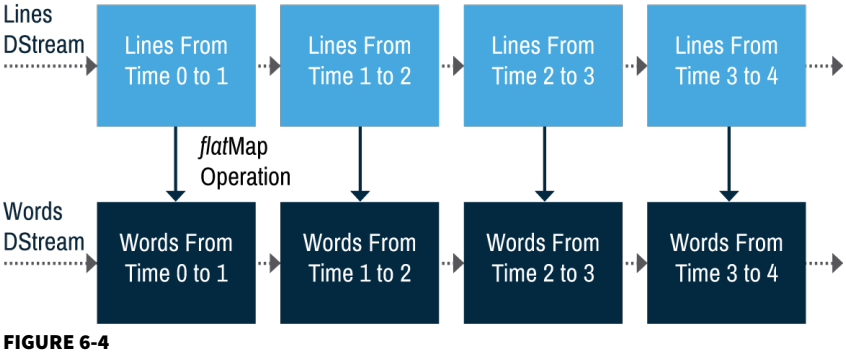

FIGURE 6-4

Extracting words from an InputStream comprising lines of text

The Spark Driver

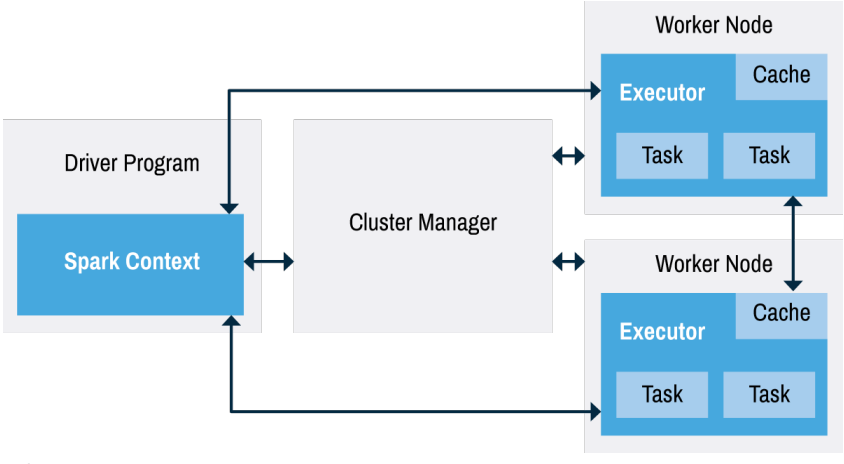

FIGURE 6-5
Components of a Spark cluster

Activities within a Spark cluster are orchestrated by a driver program using the *SparkContext*. In the case of stream-based applications, the *StreamingContext* is used. This exploits the cluster management capabilities of an external tool like Mesos or Hadoop's YARN to allocate resources to the Executor processes that actually work with data.

In a distributed and generally fault-tolerant cluster architecture, the driver is a potential point of failure, and a heavy load on cluster resources.

Particularly in the case of stream-based applications, there is an expectation and requirement that the cluster will be available and performing at all times. Potential failures in the Spark driver must therefore be mitigated, wherever possible. Spark Streaming introduced the practice of checkpointing to ensure that data and metadata associated with RDDs containing parts of a stream are routinely replicated to some form of fault-tolerant storage. This makes it feasible to recover data and restart processing in the event of a driver failure.

Processing Models

Spark Streaming itself supports commonly understood semantics for the processing of items in a data stream. These semantics ensure that the system is delivering dependable results, even in the event of individual node failures.

Items in the stream are understood to be processed in one of the following ways:

- *At most once*: Each item will either be processed once or not at all;
- *At least once*: Each item will be processed one or more times, increasing the likelihood that data will not be lost but also introducing the possibility that items may be duplicated;
- *Exactly once*: Each item will be processed exactly once.

Different input sources to Spark Streaming will offer different guarantees for the manner in which data will be processed. With version 1.3 of Spark, a new API enables *exactly once* ingest of data from Apache Kafka, improving data quality throughout the workflow. This is discussed in more detail in the **Integration Guide**.

Picking a Processing Model

From a stream processing standpoint, *at most once* is the easiest to build. This is due to the nature that the stream is "ok" with knowing that some data could be lost. Most people would think there would never be a use case which would tolerate *at most once*.

Consider a use case of a media streaming service. Lets say a customer of a movie streaming service is watching a movie and the movie player emits checkpoints every few seconds back to the media streaming service, detailing the current point in the movie. This checkpoint would be used in case the movie player crashes and the user needs to start where he/she left off. With *at most once* processing, if the checkpoint is missed, the worst case is that the user may have to rewatch a few additional seconds of the movie from the most recent checkpoint that was created. This would have a very minimal impact on a user of the system. The message might look something like:

```
{
  "checkpoint": {
    "user": "xyz123",
    "movie": "The Avengers",
    "time": "1:23:50"
  }
}
```

At least once will guarantee that none of those checkpoints will be lost. The same case with *at least once* processing would change in that the same checkpoint could potentially be replayed multiple times. If the stream processor handling a checkpoint saved the checkpoint, then crashed before it could be ac-

knowledged, the checkpoint would be replayed when the server comes back online.

This brings up an important note about replaying the same request multiple times. *At least once* guarantees every request will be processed one or more times, so there should be special considerations for creating code functions that are idempotent. This means that an action can be repeated multiple times and never produce a different result.

```
x = 4 // This is idempotent
x++ // This is NOT idempotent
```

If *at least once* was the requirement for this media streaming example, we could add a field to the checkpoint to enable a different way of acting upon the checkpoint:

```
"usersTime": "20150519T13:15:14"
```

With that extra piece of information, the function that persists the checkpoint could check to see if usersTime is less than the latest checkpoint. This would prevent overwriting a newer value and would cause the code function to be idempotent.

Within the world of streaming, it is important to understand these concepts and when they should matter for a streaming implementation.

Exactly once is the most costly model to implement. It requires special write-ahead logs to be implemented to ensure that no datum is lost and that it was acted upon exactly one-time, no-more, no-less. This model has a drastic impact on throughput and performance in a computing system because of the guarantees that it makes. *Exactly once* sounds nice and even makes people feel all warm and fuzzy because everyone understands what the end result will be. The trade-offs must be weighed before going down this route. If code functions can be made to be idempotent, then there is ***NO VALUE*** in *exactly once* processing. Generally, implementing *at least once* with idempotent functions should be the goal of any stream processing system. Functions which cannot be made to be idempotent and still require such a guarantee have little choice but to implement *exactly once* processing.

Spark Streaming vs. Others

Spark streaming operates on the concept of micro-batches. This means that Spark Streaming should not be considered a real-time stream processing engine. This is perhaps the single biggest difference between Spark Streaming and other platforms such as Apache Storm or Apache Flink.

A micro-batch consists of a series of events batched together over a period of time. The batch interval generally ranges from as little as 500ms to about 5,000ms (can be higher). While the batch interval is completely customizable, it is ***important to note*** that it is set upon the creation of the *StreamingContext*. To change this setting would require a restart of the streaming application due to it being a statically configured value.

The shorter the time frame (500ms), the closer to real time, and also the more overhead the system will endure. This is due to the need of creating more RDDs for each and every micro-batch. The inverse is also true; the longer the time frame, the further from real time and the less overhead that will occur for processing each micro-batch.

An argument often made with the concept of a micro-batch in the context of streaming applications is that it lacks the time-series data from each discrete event. This can make it more difficult to know if events arrived out of order. This may or may not be relevant to a business use case.

An application built upon Spark Streaming cannot react to every event as they occur. This is not necessarily a bad thing, but it is instead very important to make sure that everyone, everywhere understands the limitations and capabilities of Spark Streaming.

Performance Comparisons

Spark Streaming is fast, but to make comparisons between Spark Streaming and other stream processing engines, such as Apache Storm, is a difficult task. The comparisons tend not to be apples-to-apples. Most benchmarks comparisons should be taken very lightly. Because Spark Streaming is micro-batch based, it is going to tend to appear faster than nearly every system that is not micro-batch based, but this trade-off comes at the cost of latency to process the events in a stream. The closer to real-time that an event is processed, the more overhead that occurs in Spark Streaming.

Current Limitations

Two of the biggest complaints about running Spark Streaming in production are back pressure and out-of-order data.

Back pressure occurs when the volume of events coming across a stream is more than the stream processing engine can handle. There are changes that will show up in version 1.5 of Spark to enable more dynamic ingestion rate capabilities to make back pressure be less of an issue.

More work is being performed to enable user-defined time extraction functions. This will enable developers to check event time against events already processed. Work in this area is expected in a future release of Spark.

Putting Spark into Production 7

"Spark is like a fighter jet that you have to build yourself. Once you have it built though, you have a fighter jet. Pretty awesome. Now you have to learn to fly it."

This analogy came from a conversation I had with someone at Strata London in 2015. Let's break down this quote to see the value it serves in discussing Spark, and explain why this analogy may or may not be accurate.

Breaking it Down

Spark and Fighter Jets

Fighter jets are a phenomenal feat of engineering, but how is this relevant to Spark? Well, building scalable applications can be difficult. Putting them into production is even more difficult. Spark scales out of the box nearly as simply as it is to install. A lot of work has gone into the thoughts and concepts of Spark as a scalable platform.

Spark is powerful, not only in the terms of scalability, but in ease of building applications. Spark has an API that is available in multiple languages and allows nearly any business application to be built on top of it.

However, just because Spark can scale easily doesn't mean everything written to run in Spark can scale as easily.

Learning to Fly

While the API is similar or nearly identical between languages, this doesn't solve the problem of understanding the programming language of choice. While a novice programmer may be able to write Java code with minimal effort, it doesn't mean they understand the proper constructs in the language to optimize for a use case.

Let's consider analytics in Python on Spark. While a user may understand Python analytics, they may have no experience with concepts like predicate

movement, column pruning or filter scans. These features could have significant impact when running queries at scale. Here are a few other topic areas where people may overestimate how Spark works by drawing on experiences with other technologies:

- Spark supports MapReduce, but people with a lot of experience with Hadoop MapReduce might try to transfer over ideas that don't necessarily translate over to Spark, such as functional programming constructs, type safety, or lazy evaluation;
- Someone with database administration experience with any popular RDBMS system may not be thinking of partitioning and serialization in the same terms that would be useful in Spark;

Another thing that could cause problems would be trying to run multiple use cases on a single Spark cluster. Java Virtual Machine configuration settings for a Spark cluster could be optimized for a single use case. Deploying an alternate use case on the same Spark cluster may not be optimized with the same settings. However, with technologies like Mesos and YARN, this shouldn't be a real problem. Multiple Spark clusters can be deployed to cover specific use cases. It could even be beneficial to create an ETL cluster and perhaps a cluster dedicated to streaming applications, all while running on the the same underlying hardware.

While these examples are not intended to be exhaustive, they hopefully clarify the concept that any given language or platform still needs to be well understood in order to get the most from it. Thus, really learning to fly.

Assessment

This analogy is pretty good, and hopefully it doesn't scare anyone away from using Spark. The fact is that building, deploying and managing distributed systems are complicated. Even though Spark tries to simplify as much as possible with good default configuration settings, it is no exception to the level of complication that distributed systems bring.

Planning for the Coexistence of Spark and Hadoop

As discussed earlier, Spark can run on its own. It is more commonly deployed as part of a cluster, managed by Mesos or the YARN resource manager within Hadoop.

Spark should always run as close to the cluster's storage nodes as possible. Much like configuring Hadoop, network I/O is likely to be the biggest bottleneck

in a deployment. Deploying with 10Gb+ networking hardware will minimize latency and yield the best results. Never allocate more than 75% of available RAM to Spark. The operating system needs to use it as well, and going higher could cause paging. If a use case is so severely limited by 75% of available RAM, it might be time to add more servers to the cluster.

Advice and Considerations

Nearly any business out there can benefit from utilizing Spark to solve problems. Thinking through taking Spark into production is usually the tough part. Some others in the industry have been kind enough to share some ideas on how to successfully take Spark into production to solve business problems. With any luck, the information provided here will help you be more successful on your own journey to success.

Advice from our friends: Pepperdata

Reliability and Performance through Monitoring

As more organizations begin to deploy Spark in their production clusters, the need for fine-grained monitoring tools becomes paramount. Having the ability to view Spark resource consumption, and monitor how Spark applications are interacting with other workloads on your cluster, can help you save time and money by:

- troubleshooting misbehaving applications;
- monitoring and improving performance;
- viewing trend analysis over time

When deploying Spark in production, here are some crucial considerations to keep in mind:

Monitoring the Right Metrics?

How granular is your visibility into Spark's activity on your cluster? Can you view all the relevant variables you need to? These are important questions, especially for troubleshooting errant applications or behavior.

With an out-of-the-box installation, Spark's Application Web UI can display basic, per-executor information about memory, CPU, and storage. By accessing the web instance on port 4040 (default), you can see statistics about specific jobs, like their duration, number of tasks, and whether they've completed.

But this default monitoring capability isn't necessarily adequate. Take a basic scenario: suppose a Spark application is reading heavily from disk, and you want to understand how it's interacting with the file subsystem because the application is missing critical deadlines. Can you easily view detailed information about file I/O (both local file system and HDFS)? No, not with the default Spark Web UI. But this granular visibility would be necessary to see how many files are being opened con-

currently, and whether a specific disk is hot-spotting and slowing down overall performance. With the right monitoring tool, discovering that the application attempted to write heavily to disk at the same time as a MapReduce job could take seconds instead of minutes or hours using basic Linux tools like Top or Iostat.

These variables are important, and without them you may be flying blind. Having deep visibility helps you quickly troubleshoot and respond in-flight to performance issues. Invest time in researching an add-on monitoring tool for Spark that meets your organization's needs.

Is Your Monitoring Tool Intuitive?

It's great to have lots of data and metrics available to digest, but can you navigate that data quickly? Can you find what you need, and once you do, can you make sense of it? How quantitative information is displayed makes a difference. Your monitoring tool should allow you to easily navigate across different time periods, as well as to zoom in on a few seconds' worth of data. You should have the option to plot the data in various ways—line charts, by percentile, or in a stacked format, for example. Note whether you can filter the data easily by user, job, host, or queue. In short, can you use your monitoring tool intuitively, complementing your mental line of questioning? Or do you have to work around the limitations of what the tool presents?

If you have all the data, but can't sort it easily to spot trends or quickly filter it to drill down into a particular issue, then your data isn't helping you. You won't be able to effectively monitor cluster performance or take advantage of the data you do have. So make sure your tool is useful, in all senses of the word.

Can You See Global Impact?

Even if you are able to see the right metrics via an intuitive dashboard or user interface, being limited in vantage point to a single Spark job or to a single node view is not helpful. Whatever monitoring tool you choose should allow you to see not just one Spark job, but all of them—and not just all your Spark jobs, but everything else happening on your cluster, too. How is Spark impacting your HBase jobs or your other MapReduce workloads?

Spark is only one piece in your environment, so you need to know how it integrates with other aspects of your Hadoop ecosystem. This is a no-brainer from a troubleshooting perspective, but it's also a good practice for general trend analysis. Perhaps certain workloads cause greater impact to Spark performance than others, or vice versa. If you anticipate an increase in Spark usage across your organization, you'll have to plan differently than if you hadn't noticed that fact.

In summary, the reliability and performance of your Spark deployment depends on what's happening on your cluster, including both the execution of individual Spark jobs and how Spark is interacting (and impacting) your broader Hadoop environment. To understand what Spark's doing, you'll need a monitoring tool that can

provide deep, granular visibility as well as a wider, macro view of your entire system. These sorts of tools are few and far between, so choose wisely.

Spark In-Depth Use Cases 8

In a similar way to Chapter 5, we will be looking into new and exciting ways to use Spark to solve real business problems. Instead of touching on simpler examples, it is time to get into the details. These are complicated problems that are not easily solved without today's current big data technologies. It's a good thing Spark is such a capable tool set.

The first use case will show you how to build a recommendation engine. While the use case focuses on movies, recommendation engines are used all across the Internet, from applying customized labels to email, to providing a great book suggestion, to building a customized advertising engine against custom-built user profiles. Since movies are fun to watch and to talk about and are ingrained into pop culture, it only makes sense to talk about building a movie recommendation engine with Spark.

The second use case is an introduction to unsupervised anomaly detection. We will explore data for outliers with Spark. Two unsupervised approaches for detecting anomalies will be demonstrated; clustering and unsupervised random forests. The examples will use the KDD'99 dataset, commonly used for network intrusion illustrations.

The final and perhaps most intellectually stimulating use case attempts to answer a half-a-century old question, did Harper Lee write *To Kill a Mockingbird?* For many years, conspiracy buffs supported the urban legend that Truman Capote, Lee's close friend with considerably more literary creds, might have ghost-authored the novel. The author's reticence on that subject (as well as every other subject) fueled the rumors and it became another urban legend. This in-depth analysis digs into this question and uses Spark in an attempt to answer this question once and for all.

Building a Recommendation Engine with Spark

Recommendation systems help narrow your choices to those that best meet your particular needs, and they are among the most popular applications of big

data processing. This use case uses machine learning to perform parallel and iterative processing in Spark and covers:

- Collaborative filtering for recommendations with Spark
- Loading and exploring the sample data set with Spark
- Using Spark MLlib's Alternating Least Squares algorithm to make movie recommendations
- Testing the results of the recommendations

Collaborative Filtering with Spark

Collaborative filtering algorithms recommend items (this is the *filtering* part) based on preference information from many users (this is the *collaborative* part). The collaborative filtering approach is based on similarity; the basic idea is people who liked similar items in the past will like similar items in the future. In the example below, Ted likes movies A, B, and C. Carol likes movies B and C. Bob likes movie B. To recommend a movie to Bob, we calculate that users who liked B also liked C, so C is a possible recommendation for Bob. Of course, this is a tiny example. In real situations, we would have much more data to work with.

	Movie 1	Movie 2	Movie 3
Ted	4	5	5
Carol		5	5
Bob		5	?

FIGURE 8-1
Users Item Rating Matrix

Spark MLlib implements a collaborative filtering algorithm called **Alternating Least Squares (ALS)**.

ALS approximates the sparse user item rating matrix of dimension K as the product of two dense matrices--User and Item factor matrices of size U×K and I×K (see picture below). The factor matrices are also called latent feature models. The factor matrices represent hidden features which the algorithm tries to

discover. One matrix tries to describe the latent or hidden features of each user, and one tries to describe latent properties of each movie.

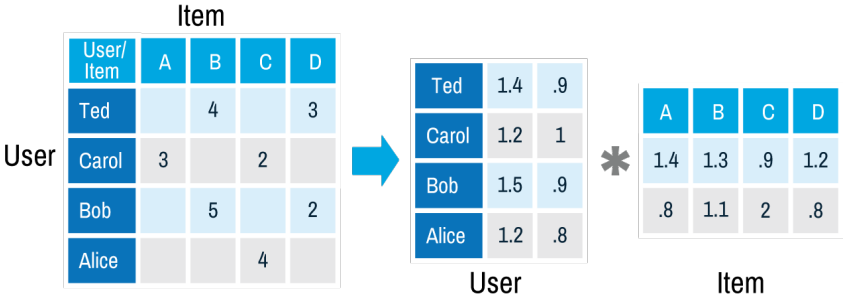

FIGURE 8-2
Calculation of a recommendation

ALS is an *iterative algorithm*. In each iteration, the algorithm alternatively fixes one factor matrix and solves for the other, and this process continues until it converges. This alternation between which matrix to optimize is where the "alternating" in the name comes from.

Typical Machine Learning Workflow

A typical machine learning workflow is shown below.

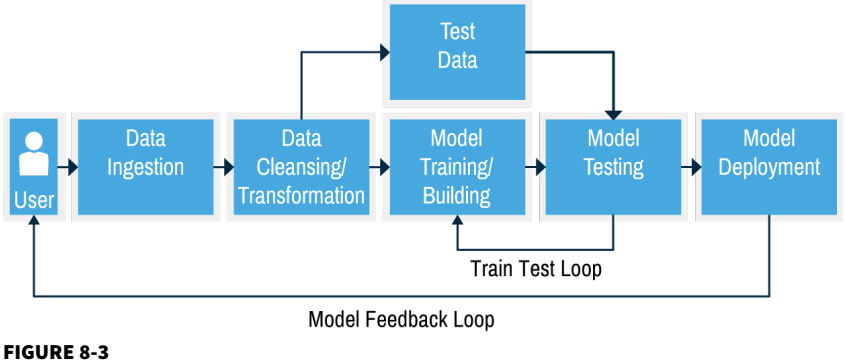

FIGURE 8-3
Maching Learning Workflow

This code will perform the following steps:

1. Load the sample data.
2. Parse the data into the input format for the ALS algorithm.
3. Split the data into two parts: one for building the model and one for testing the model.
4. Run the ALS algorithm to build/train a user product matrix model.
5. Make predictions with the training data and observe the results.
6. Test the model with the test data.

The Sample Set

The table below shows the Rating data fields with some sample data:

user id	movie id	rating
1	1193	4

The table below shows the Movie data fields with some sample data:

movie id	title	genre
1	Toy Story	animation

First, let's explore the data using Spark DataFrames with questions like:

- Count the max, min ratings along with the number of users who have rated a movie.
- Display the title for movies with ratings > 4

Loading Data into Spark DataFrames

First, we will import some packages and instantiate a sqlContext, which is the entry point for working with structured data (rows and columns) in Spark and allows for the creation of DataFrame objects.

```
// SQLContext entry point for working with structured data
val sqlContext = new org.apache.spark.sql.SQLContext(sc)

// This is used to implicitly convert an RDD to a DataFrame.
import sqlContext.implicits._
// Import Spark SQL data types
import org.apache.spark.sql._
// Import mllib recommendation data types
import org.apache.spark.mllib.recommendation.{ALS,
  MatrixFactorizationModel, Rating}
```

Below we use Scala case classes to define the Movie and User schemas corresponding to the movies.dat and users.dat files.

```
// input format MovieID::Title::Genres
case class Movie(movieId: Int, title: String, genres: Seq[String])

// input format is UserID::Gender::Age::Occupation::Zip-code
case class User(userId: Int, gender: String, age: Int,
  occupation: Int, zip: String)
```

The functions below parse a line from the movie.dat, user.dat, and rating.dat files into the corresponding Movie and User classes.

```
// function to parse input into Movie class
def parseMovie(str: String): Movie = {
      val fields = str.split("::")
      assert(fields.size == 3)
      Movie(fields(0).toInt, fields(1))
 }

// function to parse input into User class
def parseUser(str: String): User = {
      val fields = str.split("::")
      assert(fields.size == 5)
      User(fields(0).toInt, fields(1).toString, fields(2).toInt,
        fields(3).toInt, fields(4).toString)
 }
```

Below we load the data from the ratings.dat file into a Resilient Distributed Dataset (RDD). RDDs can have **transformations** and **actions**.

```
 // load the data into a RDD
val ratingText = sc.textFile("/user/user01/moviemed/ratings.dat")

// Return the first element in this RDD
ratingText.first()
```

The *first()* **action** returns the first element in the RDD, which is the String **"1::1193::5::978300760"**.

We use the org.apache.spark.mllib.recommendation.Rating class for parsing the ratings.dat file. Later we will use the Rating class as input for the ALS run method.

Then we use the map **transformation** on *ratingText*, which will apply the *parseRating* function to each element in *ratingText* and return a new RDD of Rating objects. We cache the ratings data, since we will use this data to build the matrix model. Then we get the counts for the number of ratings, movies and users.

```
// function to parse input UserID::MovieID::Rating
// Into org.apache.spark.mllib.recommendation.Rating class
def parseRating(str: String): Rating= {
      val fields = str.split("::")
      Rating(fields(0).toInt, fields(1).toInt, fields(2).toDouble)
}

// create an RDD of Ratings objects
val ratingsRDD = ratingText.map(parseRating).cache()

// count number of total ratings
val numRatings = ratingsRDD.count()

// count number of movies rated
val numMovies = ratingsRDD.map(_.product).distinct().count()

// count number of users who rated a movie
val numUsers = ratingsRDD.map(_.user).distinct().count()
```

Explore and Query with Spark DataFrames

Spark SQL provides a programming abstraction called DataFrames. A DataFrame is a distributed collection of data organized into named columns. Spark supports automatically converting an RDD containing case classes to a DataFrame with the method *toDF*, and the case class defines the schema of the table.

Below we load the data from the users and movies data files into an RDD, use the *map()* **transformation** with the parse functions, and then call *toDF()* which returns a DataFrame for the RDD. Then we register the DataFrames as temp tables so that we can use the tables in SQL statements.

```
// load the data into DataFrames
val usersDF = sc.textFile("/user/user01/moviemed/users.dat")
  .map(parseUser).toDF()
val moviesDF = sc.textFile("/user/user01/moviemed/movies.dat")
  .map(parseMovie).toDF()

// create a DataFrame from the ratingsRDD
val ratingsDF = ratingsRDD.toDF()

// register the DataFrames as a temp table
ratingsDF.registerTempTable("ratings")
moviesDF.registerTempTable("movies")
usersDF.registerTempTable("users")
```

DataFrame *printSchema()* prints the schema to the console in a tree format.

```
usersDF.printSchema()

moviesDF.printSchema()

ratingsDF.printSchema()
```

Here are some example queries using Spark SQL with DataFrames on the Movie Lens data. The first query gets the maximum and minimum ratings along with the count of users who have rated a movie.

```
// Get the max, min ratings along with the count of users who have
// rated a movie.
val results = sqlContext.sql(
  "select movies.title, movierates.maxr, movierates.minr, movie-
rates.cntu
    from(SELECT ratings.product, max(ratings.rating) as maxr,
    min(ratings.rating) as minr,count(distinct user) as cntu
    FROM ratings group by ratings.product ) movierates
    join movies on movierates.product=movies.movieId
    order by movierates.cntu desc")

// DataFrame show() displays the top 20 rows in  tabular form
results.show()
```

The query below finds the users who rated the most movies, then finds which movies the most active user rated higher than 4. We will get recommendations for this user later.

```
// Show the top 10 most-active users and how many times they rated
// a movie
val mostActiveUsersSchemaRDD = sqlContext.sql(
  "SELECT ratings.user, count(*) as ct from ratings group by
  ratings.user order by ct desc limit 10")

println(mostActiveUsersSchemaRDD.collect().mkString("\n"))

// Find the movies that user 4169 rated higher than 4
val results = sqlContext.sql("SELECT ratings.user, ratings.prod-
uct,
  ratings.rating, movies.title FROM ratings JOIN movies
  ON movies.movieId=ratings.product
  where ratings.user=4169 and ratings.rating > 4")

results.show
```

Using ALS with the Movie Ratings Data

Now we will use the MLlib ALS algorithm to learn the latent factors that can be used to predict missing entries in the user-item association matrix. First we separate the ratings data into training data (80%) and test data (20%). We will get recommendations for the training data, and then we will evaluate the predictions with the test data. This process of taking a subset of the data to build the model and then verifying the model with the remaining data is known as cross validation; the goal is to estimate how accurately a predictive model will perform in practice. To improve the model, this process is often done multiple times with different subsets; we will only do it once.

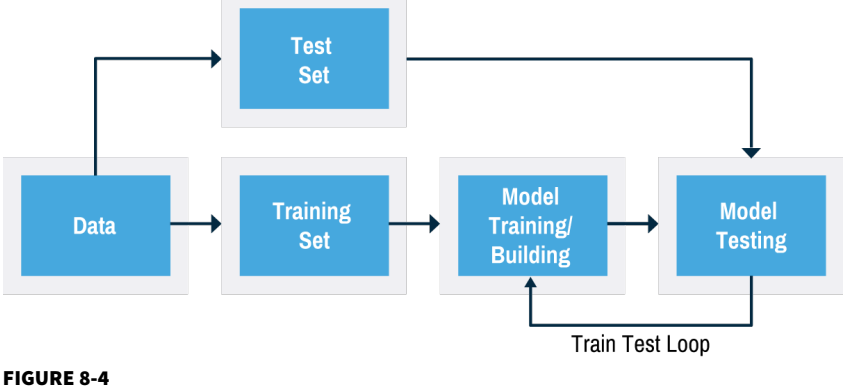

FIGURE 8-4
Training Loop

We run ALS on the input trainingRDD of Rating(user, product, rating) objects with the rank and Iterations parameters:

- rank is the number of latent factors in the model.
- iterations is the number of iterations to run.

The ALS run(trainingRDD) method will build and return a MatrixFactorizationModel, which can be used to make product predictions for users.

```
// Randomly split ratings RDD into training
// data RDD (80%) and test data RDD (20%)
val splits = ratingsRDD.randomSplit(Array(0.8, 0.2), 0L)

val trainingRatingsRDD = splits(0).cache()
val testRatingsRDD = splits(1).cache()

val numTraining = trainingRatingsRDD.count()
val numTest = testRatingsRDD.count()
println(s"Training: $numTraining, test: $numTest.")

// build a ALS user product matrix model with rank=20, itera-
tions=10
val model = (new ALS().setRank(20).setIterations(10)
  .run(trainingRatingsRDD))
```

Making Predictions

Now we can use the MatrixFactorizationModel to make predictions. First, we will get movie predictions for the most active user, 4169, with the recommendProducts() method, which takes as input the userid and the number of products to recommend. Then we print out the recommended movie titles.

```
// Get the top 4 movie predictions for user 4169
val topRecsForUser = model.recommendProducts(4169, 5)

// get movie titles to show with recommendations
val movieTitles=moviesDF.map(array => (array(0), array(1)))
  .collectAsMap()

// print out top recommendations for user 4169 with titles
topRecsForUser.map(rating => (movieTitles(
  rating.product), rating.rating)).foreach(println)
```

Evaluating the Model

Next, we will compare predictions from the model with actual ratings in the *testRatingsRDD*. First we get the user product pairs from the *testRatingsRDD* to pass to the MatrixFactorizationModel *predict*(user: Int, product: Int) method, which will return predictions as *Rating*(user, product, rating) objects.

```
// get user product pair from testRatings
val testUserProductRDD = testRatingsRDD.map {
  case Rating(user, product, rating) => (user, product)
}

// get predicted ratings to compare to test ratings
val predictionsForTestRDD  = model.predict(testUserProductRDD)

predictionsForTestRDD.take(10).mkString("\n")
```

Now we will compare the test predictions to the actual test ratings. First we put the predictions and the test RDDs in this key, value pair format for joining: ((user, product), rating). Then we print out the (user, product), (test rating, predicted rating) for comparison.

```
// prepare predictions for comparison
val predictionsKeyedByUserProductRDD = predictionsForTestRDD.map{
  case Rating(user, product, rating) => ((user, product), rating)
}

// prepare test for comparison
val testKeyedByUserProductRDD = testRatingsRDD.map{
  case Rating(user, product, rating) => ((user, product), rating)
}

//Join the test with predictions
val testAndPredictionsJoinedRDD = testKeyedByUserProductRDD
  .join(predictionsKeyedByUserProductRDD)

// print the (user, product),( test rating, predicted rating)
testAndPredictionsJoinedRDD.take(3).mkString("\n")
```

The example below finds false positives by finding predicted ratings which were >= 4 when the actual test rating was <= 1. There were 557 false positives out of 199,507 test ratings.

```
val falsePositives = (testAndPredictionsJoinedRDD
  .filter{case ((user, product), (ratingT, ratingP)) =>
    (ratingT <= 1 && ratingP >=4)})
falsePositives.take(2)

falsePositives.count
```

Next we evaluate the model using Mean Absolute Error (MAE). MAE is the absolute differences between the predicted and actual targets.

```
// Evaluate the model using Mean Absolute Error (MAE) between test
// and predictions
val meanAbsoluteError = testAndPredictionsJoinedRDD.map {
  case ((user, product), (testRating, predRating)) =>
    val err = (testRating - predRating)
    Math.abs(err)
}.mean()
```

Use case provided by: Dan Mallinger, Director of Data Science at Think Big, a Teradata company

Unsupervised Anomaly Detection with Spark

The term anomalous data refers to data that are different from what are expected or normally occur. Detecting anomalies is important in most industries. For example, in network security, anomalous packets or requests can be flagged as errors or potential attacks. In customer security, anomalous online behavior can be used to identify fraud. And in manufacturing and the Internet of Things, anomaly detection is useful for identifying machine failures.

Anomaly detection flags "bad" data—whether that refers to data quality, malicious action, or simply user error. And because bad data are typically—and hopefully—rare, modeling bad data can be difficult when data are sampled. But the act of sampling eliminates too many or all of the anomalies needed to build a detection engine. Instead, you want large data sets—with all their data quality issues—on an analytics platform that can efficiently run detection algorithms. Apache Spark, as a parallelized big data tool, is a perfect match for the task of anomaly detection.

By framing anomalies as "bad data," it becomes clear that the patterns of what we call "bad data" change over time. It's an arms race: the act of fixing an error will likely result in new errors and stopping one type of attack may lead malicious actors to devise new attacks. Because of an ever-changing environment, detection should not be fixated on one anomalous pattern. What is required then is an unsupervised approach (i.e., one that does not require training data with records flagged as anomalous or not).

Getting Started

Using the KDD'99 data set, we will filter out a number of columns for two reasons: ease of example and removal of labeled data, as this use case only demonstrates unsupervised approaches.

```
import numpy as np
INPUT = "hdfs://localhost/data/kdd"

def parse_line(line):
  bits = line.split(",")
  return np.array([float(e) for e in bits[4:12]])

df = sc.textFile(INPUT).map(parse_line)
```

Identifying Univariate Outliers

To begin, we want to get summary statistics for columns in our data. The *stats()* function provides these summaries. We can then use the results to get all records where the first column lies more than two standard deviations from the mean:

```
stats = df.map(lambda e: e[0]).stats()
mean, stdev = stats.mean(), stats.stdev()
outliers = df.filter(lambda e: not (mean - 2 * stdev > e[0] >
mean + 2 * stdev))
outliers.collect()
```

Unfortunately, this approach has two limitations. First, it requires knowledge of the parameter distribution: in this case, it assumes data follow a roughly normal distribution. Second, in this example we can only determine outliers by looking at individual variables, but anomalous data is often defined by the relationship between variables. For example, going to the money transfer page of a bank is normal, but doing so without first visiting an account page can signal a fraudulent agent. Thus, we should move ahead with a multivariate approach instead.

Detection with Clustering

Clustering refers to an unsupervised approach whereby data points close to one another are grouped together. A common approach called k-means will produce k clusters where the distance between points is calculated using Euclidean distance. This approach can be quickly run in Spark via MLlib:

```
from pyspark.mllib.clustering import KMeans
clusters = KMeans.train(df, 5, maxIterations=10,
    runs=1, initializationMode="random")
```

There are now five clusters created. We can see the size of each cluster by labeling each point and counting. However, note that determining the optimal number of

clusters requires iteratively building a model, omitted here because we move to an alternative approach in the next section.

```
cluster_sizes = df.map(lambda e: clusters.predict(e)).countByVal-
ue()
```

K-means is sensitive to outliers. This means that anomalous data often ends up in clusters alone. Looking at the *cluster_sizes* dictionary, we find clusters 0 and 2 have 1 and 23 data points each, suggesting they are anomalous. But we cannot be certain without further inspection.

However, because we have a small number of clusters (and probably too few), we may find anomalous data within non-anomalous clusters. A first inspection should look at the distribution of distances of data points from cluster centers:

```
def get_distance(clusters):
  def get_distance_map(record):
    cluster = clusters.predict(record)
    centroid = clusters.centers[cluster]
    dist = np.linalg.norm(record - centroid)
    return (dist, record)
  return get_distance_map

data_distance = df.map(get_distance(clusters))
hist = data_distance.keys().histogram(10)
```

Looking at the histogram (represented as a Python list) shows that occurrences of values decline as the distance grows, with the exception of a spike in the last bucket. Those data points are a good starting place to seek anomalies.

K-means clustering is a powerful tool. However, it does have a number of challenges. First, selecting the appropriate value for k can be difficult. Second, K-means is sensitive to the scale of variables (see Exercise 2). Last, there are no thresholds or scores that can be readily used to evaluate new data as anomalous. Because of these limitations, let us move on to an alternative approach.

Exercise 1: Run these calculations and explore the data points. Can you explain the bump at the end of the histogram?

Exercise 2: K-means is sensitive to the scale of variables. This means variables with larger variance will overtake your calculations. Try standardizing variables by subtracting the mean and dividing by the standard deviation for each. How does this change your results?

Detection with Unsupervised Random Forests

Random forests are a powerful prediction method that uses collections of trees with a random parameter holdout to build models that often outperform individual decision trees. However, the random forest is normally a supervised approach, requiring labeled data.

In this section, we introduce a method for turning a supervised model into an unsupervised model for anomaly detection. Unsupervised random forests have a number of advantages over k-means for simple detection. First, they are less sensitive to variable scale. Second, they can fit to "normal" behavior and thus can provide a probability of a data point being anomalous.

To create an unsupervised model from a supervised one, we create a new dataset of "dummy" data. We create dummy data by sampling from within columns of the original data and joining those columns. This creates a dataset whose column values are non-anomalous, but whose relationships between columns are. The unisample function provides an example:

```
def unisample(df, fraction=1.0):
  columns = df.first()
  new_df = None
  for i in range(0, len(columns)):
    column = df.sample(withReplacement=True, fraction=fraction) \
            .map(lambda row: row[i]) \
            .zipWithIndex() \
            .map(lambda e: (e[1], [e[0]]))
    if new_df is None:
      new_df = column
    else:
      new_df = new_df.join(column)
      new_df = new_df.map(lambda e: (e[0], e[1][0] + e[1][1]))
  return new_df.map(lambda e: e[1])
```

Next we label the dummy data 0 (for "not real") and original data 1 (for "real"). The *supervised2unsupervised* function shows how to create such a dataset. It also takes a supervised model and returns an unsupervised model, which we will do next:

```
def supervised2unsupervised(model):
  def run(df, *args, **kwargs):
    unisampled_df = unisample(df)
    unisampled_df = unisample(df)
    labeled_data = df.map(lambda e: LabeledPoint(1, e)) \
             .union(unisampled_df.map(lambda e: LabeledPoint(0,
e)))
    return model(labeled_data, *args, **kwargs)
  return run
```

Now we can create and fit an unsupervised random forest using the RandomForest function provided by MLlib:

```
from pyspark.mllib.regression import LabeledPoint
from pyspark.mllib.tree import RandomForest

unsupervised_forest = supervised2unsupervised(RandomForest.train-
Classifier)
rf_model = unsupervised_forest(df, numClasses=2, categoricalFea-
turesInfo={},
                  numTrees=10, featureSubsetStrategy="auto",
                  impurity='gini', maxDepth=15, maxBins=50)
```

We have now created a new model (*rf_model*) that is trained to recognize "normal" data, giving us a robust anomaly detection tool without requiring labeled data. And anytime we run *rf_model.predict*, we can get a probability of a data point being anomalous!

Exercise 3: The forest in this example has suboptimal parameters (e.g., only have 10 trees). Reconfigure the forest to improve the overall prediction accuracy of non-anomalous data.

Exercise 4: This approach is sensitive to the dummy data generated. Either by "uni-sampling" with subsets or other approaches, try and bootstrap many forests. Create a single *predict* function that returns the mean and standard deviation across this collection of models. Use this to generate confidence intervals over your predictions.

Machine Learning Library (MLlib) with Spark

Did Harper Lee write *To Kill a Mockingbird?* The recent *'discovery'* and subsequent publication of Harper Lee's earlier novel *Go Set a Watchmen* has generated renewed scrutiny of the chain of events. Is the newly published book a discarded rough draft that was to become the universally beloved classic, or was it a truly forgotten separate work that deserves to be cast in the literary limelight for analysis? A concise summary of the publishing controversy was written in an **op-ed column** by the New York Times.

The new book offers curious readers an opportunity to analyze the two works together with machine learning tools that are ideal for classifying text among a corpus of documents. Apache Spark has a mature set of libraries for text-based analysis that can be leveraged with very few lines of code.

The publisher of *Go Set a Watchman* is unlikely to make available their best seller even for lofty academic purposes. Luckily, the Wall Street Journal printed the **first chapter** on July 10, 2015 for anyone to analyze. In this use case, features will be extracted from the first chapter of each book, and then a model

will be built to show the difference between them. Comparing passages from each may provide clues as to the authorship.

Dissecting a Classic by the Numbers

The theory behind document classification is that text from the same source will contain similar combinations of words with comparable frequency. Any conclusions based from this type of analysis are only as strong as that assumption.

To build a model to classify documents, text must be translated into numbers. This involves standardizing the text, converting to numbers (via hashing), and then adjusting the word importance based on its relative frequency.

Text standardization was done with Apache Lucene. An example below shows how to perform this with the Spark shell:

```
./bin/spark-shell --packages \
   "org.apache.lucene:lucene-analyzers-common:5.1.0"
val line="Flick. A tiny, almost invisible movement, " +
  "and the house was still."
val tokens=Stemmer.tokenize(line)
# tokens: Seq[String] = ArrayBuffer(flick, tini, almost,
#   invis, movement, hous, still)
```

The Stemmer object that invokes the Lucene analyzer comes from an article on **classifying documents using Naive Bayes on Apache Spark / MLlib**. Notice how the line describing the tranquility of the Radley house is affected. The punctuation and capitalization is removed, and words like “house” are stemmed, so tokens with the same root (“housing”, “housed”, etc.) will be considered equal. Next, we translate those tokens into numbers and count how often they appear in each line. Spark’s HashingTF library performs both operations simultaneously.

```
import org.apache.spark.mllib.feature.HashingTF
val tf = new HashingTF(10)

val hashed = tf.transform(tokens)
```

A “hash” is a one-way translation from text to an integer (i.e. once it’s translated, there’s no way to go back). Initializing the hash with HashingTF(10) notifies Spark that we want every string mapped to the integers 0-9. The transform method performs the hash on each word, and then provides the frequency count for each. This is an impractical illustration, and would result in a huge number of “collisions” (different strings assigned the same number).

The default size of the resulting vector of token frequencies is 1,000,000. The size and number of collisions are inversely related. But a large hash also re-

quires more memory. If your corpus contains millions of documents, this is an important factor to consider. For this analysis, a hash size of 10,000 was used.

The last step in the text preparation process is to account for the rareness of words--we want to reward uncommon words such as “chifferobe” with more importance than frequent words such as “house” or “brother”. This is referred to as TF-IDF transformation and is available as an (almost) one-liner in Spark.

```
import org.apache.spark.mllib.feature.IDF
val idfModel = new IDF(minDocFreq = 3).fit(trainDocs)
val idfs = idfModel.transform(hashed)
```

The “fit” method of the IDF library examines the entire corpus to tabulate the document count for each word. On the second pass, Spark creates the TF-IDF for each non-zero element (tokeni) as the following:

$$TFIDF_i = \sqrt{TF} * \ln(doc\ count + 1/doc\ count_i + 1)$$

FIGURE 8-5
The “fit” method equation

A corpus of many documents is needed to create an IDF dictionary, so in the example above, excerpts from both novels were fed into the fit method. The transform method was then used to convert individual passages to TF-IDF vectors.

Having been transformed into TF-IDF vectors, passages from both books are now ready to be classified.

Building the Classifier

The secret to getting value from business problems is not the classification; it is primarily about ranking objects based on the confidence of our decision and then leveraging the value of a good decision minus the cost of a misidentification. Spark has several machine learning algorithms that are appropriate for this task.

During examination of the text, it was noted that a few modifications should be made to the novels to make the comparison more “fair.” *To Kill a Mockingbird* was written in the first person and includes many pronouns that would be giveaways (e.g., “I”, “our”, “my”, “we”, etc.). These were removed from both books.

Due to the inevitability of variable sentence length in novels, passages were created as a series of ten consecutive words.

The parsed passages were combined, split into training and testing sets, and then transformed with the idfModel built on the training data using the code below:

```
val data = mockData.union(watchData)
val splits = data.randomSplit(Array(0.7, 0.3))
val trainDocs = splits(0).map{ x=>x.features}
val idfModel = new IDF(minDocFreq = 3).fit(trainDocs)
val train = splits(0).map{
  point=>LabeledPoint(point.label,idfModel.transform(point.fea-
tures))
}
val test = splits(1).map{
  point=>LabeledPoint(point.label,idfModel.transform(point.fea-
tures))
}
train.cache()
```

Using randomly split data files for training and testing a model is standard procedure for insuring performance is not a result of over-training (i.e., memorizing the specific examples instead of abstracting the true patterns). It is critical that the idfModel is built only on the training data. Failure to do so may result in overstating your performance on the test data.

The data are prepared for machine learning algorithms in Spark. Naive Bayes is a reasonable first choice for document classification. The code below shows the training and evaluation of a Naive Bayes model on the passages.

```
import org.apache.spark.mllib.classification.{NaiveBayes,
  NaiveBayesModel}
val nbmodel = NaiveBayes.train(train, lambda = 1.0)
val bayesTrain = train.map(p => (nbmodel.predict(p.features),
p.label))
val bayesTest = test.map(p => (nbmodel.predict(p.features), p.la-
bel))
println("Mean Naive Bayes performance")
(bayesTrain.filter(x => x._1 == x._2).count() /
  bayesTrain.count().toDouble,
  bayesTest.filter(x => x._1 == x._2).count() /
  bayesTest.count().toDouble)
```

Applying the Naive Bayes algorithm in Spark gives a classification from which accuracy and a confusion matrix can be derived. The method makes the correct classification on 90.5% of the train records and 70.7% of the test records (performance on the training is almost always better than the test). The confusion matrix on the test data appears below:

	Correct Label	
Classified as	Mockingbird	Watchman
Mockingbird	82	41
Watchman	10	41

FIGURE 8-6
Naive Bayes Confusion Matrix on test data

The diagonal elements of the confusion matrix represent correct classifications and the off-diagonal counts are classification errors. It is informative to look at a confusion matrix (especially when there are more than two classes). The better the classification rate on the test set, the more separable the populations. However, when data scientists are looking to apply classification to a business problem, they prefer to examine how well the algorithm rank-orders the results.

Currently, Spark does not support a user-supplied threshold for Naive Bayes. Only the best classification rate in the training data is reported. But in real business problems, there is an overhead associated with a misclassification so that the "best" rate may not be the optimal rate. It is of keen interest to the business to find the point at which maximum value of correct classifications is realized when accounting for incorrect answers. To do this via Spark, we need to use methods that allow for analysis of the threshold.

Given the number of features (a TF-IDF vector of size 10,000) and the nature of the data, Spark's tree-based ensemble methods are appropriate. Both Random Forest and Gradient Boosted Trees are available.

```
import org.apache.spark.mllib.tree.RandomForest
import org.apache.spark.mllib.tree.model.RandomForestModel
import org.apache.spark.mllib.tree.GradientBoostedTrees
import org.apache.spark.mllib.tree.configuration.BoostingStrategy
import org.apache.spark.mllib.tree.model.GradientBoostedTreesModel

// RANDOM FOREST REGRESSION
val categoricalFeaturesInfo = Map[Int, Int]()
val numClasses = 2
val featureSubsetStrategy = "auto"
val impurity = "variance"
val maxDepth = 10
val maxBins = 32
val numTrees = 50
val modelRF = RandomForest.trainRegressor(train,
  categoricalFeaturesInfo, numTrees, featureSubsetStrategy,
  impurity, maxDepth, maxBins)

// GRADIENT BOOSTED TREES REGRESSION
val boostingStrategy = BoostingStrategy.defaultParams("Regres-
sion")
boostingStrategy.numIterations = 50
boostingStrategy.treeStrategy.maxDepth = 5
boostingStrategy.treeStrategy.categoricalFeaturesInfo = Map[Int,
Int]()
val modelGB = GradientBoostedTrees.train(train, boostingStrategy)
```

The regression model options (estimating vs. classifying) will produce continuous outputs that can be used to find the right threshold. Both of these methods can be configured with tree depth and number of trees. Read the Spark documentation for details, but the general rules of thumb are the following:

- Random Forest: trees are built in parallel and overtraining decreases with more trees, so setting this number to be large is a great way to leverage a Hadoop environment. The max depth should be larger than GBT.
- Gradient Boosted Trees: the number of trees is directly related to overtraining, and the trees are not built in parallel. This method can produce some extremely high classification rates on the training data, but set the max depth of trees to be smaller than random forest.

The table below shows the commands to calculate the ROC (Receiver Operating Characteristic) for the Random Forest model--the ROC will tell the real story on the model performance.

```
//// Random forest model metrics on training data
val trainScores = train.map { point =>
  val prediction = modelRF.predict(point.features)
  (prediction, point.label)

//// Random forest model metrics on training data
val trainScores = train.map { point =>
  val prediction = modelRF.predict(point.features)
  (prediction, point.label)
}
val metricsTrain = new BinaryClassificationMetrics(trainScores,
100)
val trainroc= metricsTrain.roc()
trainroc.saveAsTextFile("/ROC/rftrain")
metricsTrain.areaUnderROC()
```

These are the training results.

```
//// Random forest model metrics on test data
val testScores = test.map { point =>
  val prediction = modelRF.predict(point.features)
  (prediction, point.label)
}
val metricsTest = new BinaryClassificationMetrics(testScores,100)
val testroc= metricsTest.roc()
testroc.saveAsTextFile("/ROC/rftest")
metricsTest.areaUnderROC()
```

To calculate an ROC, the following steps are performed:

1. Results are binned according to score (highest to lowest).
2. In each bin, the number of each class is tabulated (Mockingbird vs. Watchman passages).
3. Starting with the highest bin, generate a data point containing the cumulative percent of the total Mockingbird and Watchman passages that have occurred.
4. Graphing those points for the Random Forest and Gradient Boosted Trees yields the following curves:

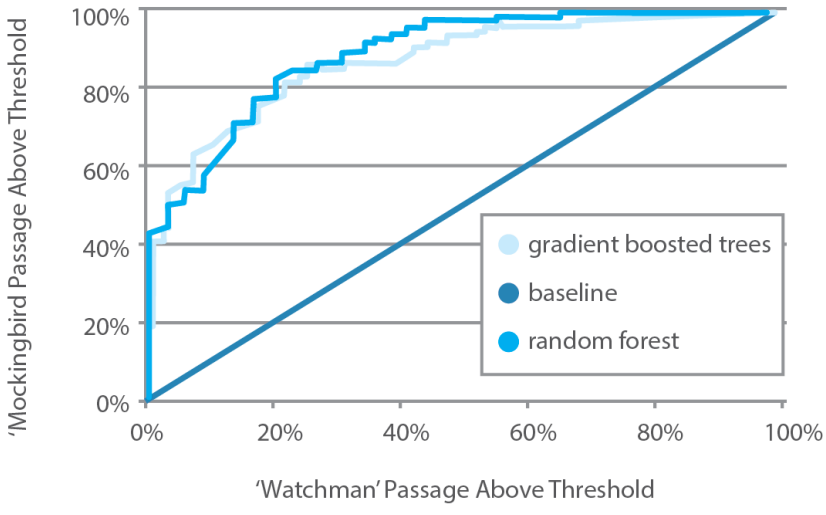

FIGURE 8-7
ROC for test data by algorithm

The diagonal "baseline" is the performance one could expect from random guessing (i.e., selecting 50% of the passages, you would expect to find half of each book's examples). Any performance better than that is considered the "lift" delivered by the model. It should be intuitive from examining the graph that steeper, higher curves provide greater lift. The table below quantifies the area under the ROC, which is a standard metric used by data scientists to evaluate the performance of many models simultaneously.

Algorithm	Training Data	Test Data
Random Forest	0.981	0.884
Gradient Boosted Trees	0.999	0.867

FIGURE 8-8
Area under the ROC by algorithm

The Gradient Boosted Tree model achieved an essentially perfect 1.0 area under the curve. This implies that the model scored all Mockingbird passages higher than all Watchman passages. However, the Random Forest model has

higher performance on the test set (0.884 vs 0.867), so it is assumed to generalize better.

In the setting of a business problem, the underlying data of the ROC is used to estimate how many items of interest can be identified when the real cost of an error is considered. Focusing on the highest scoring items from the model and working down the list is where real value comes from.

The results cannot be interpreted as conclusive, but there is significant lift displayed on these curves, and that doesn't look good for Harper Lee.

The Verdict

There are plenty of great tools to build classification models. Apache Spark provides an excellent framework for building solutions to business problems that can extract value from massive, distributed datasets.

Machine learning algorithms cannot answer the great mysteries of life. But they do provide evidence for humans to consider when interpreting results, assuming the right question is asked in the first place.

Readers are encouraged to check out these books themselves and reach their own conclusions. If the controversy surrounding the publication of Harper Lee's books causes more people to read them, that's probably a good thing.

All of the data and code to train the models and make your own conclusions using Apache Spark are located in **github**.

Getting Started with Apache Spark Conclusion

We have covered a lot of ground in this book. This is by no means everything to be experienced with Spark. Spark is constantly growing and adding new great functionality to make programming with it easier. Projects integrating with Spark seem to pop up almost daily. The **Apache Mahout** project has done some integration to bring more machine learning libraries to Spark. Projects **Jupyter** and **Apache Zeppelin** bring Spark to web notebooks.

This book will continue to evolve as things change or important new content is created within Spark. Bookmark things that you feel are important to you. If you didn't work through all the examples, come back and reference them later when you are ready.

There is also an *Apache Spark Cheat Sheet* available in **HTML**. This cheat sheet covers the most commonly referenced functionality, including links to the latest API documentation for each method covered. Hopefully this will help you become the Spark expert you want to become in no time at all.

Finally, if you haven't already, I suggest heading over and trying out the **MapR Sandbox**. It is a simple way to get running with Spark. The MapR Sand-

box provides tutorials, demo applications, and browser-based user interfaces to let you get started quickly with Spark and Hadoop.

Good luck to you in your journey with Apache Spark.

Apache Spark Developer Cheat Sheet 9

Transformations (return new RDDs – Lazy)

Where	Function	DStream API	Description
RDD	map(function)	Yes	Return a new distributed dataset formed by passing each element of the source through a function.
RDD	filter(function)	Yes	Return a new dataset formed by selecting those elements of the source on which function returns true.
OrderedRDD Functions	filterByRange(lower, upper)	No	Returns an RDD containing only the elements in the the inclusive range lower to upper.
RDD	flatMap(function)	Yes	Similar to map, but each input item can be mapped to 0 or more output items (so function should return a Seq rather than a single item).
RDD	mapPartitions(function)	Yes	Similar to map, but runs separately on each partition of the RDD.
RDD	mapPartitionsWithIndex(function)	No	Similar to mapPartitions, but also provides function with an integer value representing the index of the partition.
RDD	sample(withReplacement, fraction, seed)	No	Sample a fraction of the data, with or without replacement, using a given random number generator seed.

Where	Function	DStream API	Description
RDD	union(otherDataset)	Yes	Return a new dataset that contains the union of the elements in the datasets.
RDD	intersection(otherDataset)	No	Return a new RDD that contains the intersection of elements in the datasets.
RDD	distinct([numTasks])	No	Return a new dataset that contains the distinct elements of the source dataset.
PairRDD Functions	groupByKey([numTasks])	Yes	Returns a dataset of (K, Iterable<V>) pairs. Use reduceByKey or aggregateByKey to perform an aggregation (such as a sum or average).
PairRDD Functions	reduceByKey(function, [numTasks])	Yes	Returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function.
PairRDD Functions	aggregateByKey(zeroValue)(seqOp, combOp, [numTasks])	No	Returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type.
OrderedRDD Functions	sortByKey([ascending], [numTasks])	No	Returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument.
PairRDD Functions	join(otherDataset, [numTasks])	Yes	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through leftOuterJoin, rightOuterJoin, and fullOuterJoin.
PairRDD Functions	cogroup(otherDataset, [numTasks])	Yes	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (Iterable<V>, Iterable<W>)) tuples.
RDD	cartesian(otherDataset)	No	When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements).

Where	Function	DStream API	Description
RDD	**pipe(command, [envVars])**	No	Pipe each partition of the RDD through a shell command, e.g. a Perl or bash script.
RDD	**coalesce(numPartitions)**	No	Decrease the number of partitions in the RDD to numPartitions. Useful for running operations more efficiently after filtering down a large dataset.
RDD	**repartition(numPartitions)**	**Yes**	Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network.
OrderedRDD Functions	**repartitionAndSortWithinPartitions(partitioner)**	No	Repartition the RDD according to the given partitioner and, within each resulting partition, sort records by their keys. More efficient than calling repartition and then sorting.

Actions (return values – NOT Lazy)

Where	Function	DStream API	Description
RDD	reduce(function)	Yes	Aggregate the elements of the dataset using a function (which takes two arguments and returns one).
RDD	collect()	No	Return all the elements of the dataset as an array at the driver program. Best used on sufficiently small subsets of data.
RDD	count()	Yes	Return the number of elements in the dataset.
RDD	countByValue()	Yes	Return the count of each unique value in this RDD as a local map of (value, count) pairs.
RDD	first()	No	Return the first element of the dataset (similar to take(1)).
RDD	take(n)	No	Return an array with the first n elements of the dataset.
RDD	takeSample(withReplacement, num, [seed])	No	Return an array with a random sample of num elements of the dataset.
RDD	takeOrdered(n, [ordering])	No	Return the first n elements of the RDD using either their natural order or a custom comparator.
RDD	saveAsTextFile(path)	Yes	Write the elements of the dataset as a text. Spark will call toString on each element to convert it to a line of text in the file.
SequenceFileRDD Functions	saveAsSequenceFile(path) (Java and Scala)	No	Write the elements of the dataset as a Hadoop SequenceFile in a given path. For RDDs of key-value pairs that use Hadoop's Writable interface.
RDD	saveAsObjectFile(path) (Java and Scala)	Yes	Write the elements of the dataset in a simple format using Java serialization, which can then be loaded using SparkContext.objectFile().

Where	Function	DStream API	Description
PairRDD Functions	**countByKey()**	No	Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.
RDD	**foreach(function)**	**Yes**	Run a function on each element of the dataset. This is usually done for side effects such as updating an Accumulator.

Persistence Methods

Where	Function	DStream API	Description
RDD	cache()	Yes	Don't be afraid to call cache on RDDs to avoid unnecessary recomputation. NOTE: This is the same as persist(MEMORY_ONLY).
RDD	persist([Storage Level])	Yes	Persist this RDD with the default storage level.
RDD	unpersist()	No	Mark the RDD as non-persistent, and remove its blocks from memory and disk.
RDD	checkpoint()	Yes	Save to a file inside the checkpoint directory and all references to its parent RDDs will be removed.

Additional Transformation and Actions

Where	Function	Description
SparkContext	doubleRDDToDoubleRDDFunctions	Extra functions available on RDDs of Doubles
SparkContext	numericRDDToDoubleRDDFunctions	Extra functions available on RDDs of Doubles
SparkContext	rddToPairRDDFunctions	Extra functions available on RDDs of (key, value) pairs
SparkContext	hadoopFile()	Get an RDD for a Hadoop file with an arbitrary InputFormat
SparkContext	hadoopRDD()	Get an RDD for a Hadoop file with an arbitrary InputFormat
SparkContext	makeRDD()	Distribute a local Scala collection to form an RDD
SparkContext	parallelize()	Distribute a local Scala collection to form an RDD
SparkContext	textFile()	Read a text file from a file system URI
SparkContext	wholeTextFiles()	Read a directory of text files from a file system URI

Extended RDDs w/ Custom Transformations and Actions

RDD Name	Description
CoGroupedRDD	A RDD that cogroups its parents. For each key k in parent RDDs, the resulting RDD contains a tuple with the list of values for that key.
EdgeRDD	Storing the edges in columnar format on each partition for performance. It may additionally store the vertex attributes associated with each edge.
JdbcRDD	An RDD that executes an SQL query on a JDBC connection and reads results. For usage example, see test case JdbcRDDSuite.
ShuffledRDD	The resulting RDD from a shuffle.
VertexRDD	Ensures that there is only one entry for each vertex and by pre-indexing the entries for fast, efficient joins.

Streaming Transformations

Where	Function	Description
DStream	window(windowLength, slideInterval)	Return a new DStream which is computed based on windowed batches of the source DStream.
DStream	countByWindow(windowLength, slideInterval)	Return a sliding window count of elements in the stream.
DStream	reduceByWindow(function, windowLength, slideInterval)	Return a new single-element stream, created by aggregating elements in the stream over a sliding interval using function.
PairDStream Functions	reduceByKeyAndWindow(function, windowLength, slideInterval, [numTasks])	Returns a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function over batches in a sliding window.
PairDStream Functions	reduceByKeyAndWindow(function, invFunc, windowLength, slideInterval, [numTasks])	A more efficient version of the above reduceByKeyAndWindow(). Only applicable to those reduce functions which have a corresponding “inverse reduce” function. Checkpointing must be enabled for using this operation.
DStream	countByValueAndWindow(windowLength, slideInterval, [numTasks])	Returns a new DStream of (K, Long) pairs where the value of each key is its frequency within a sliding window.
DStream	transform(function)	The transform operation (along with its variations like transformWith) allows arbitrary RDD-to-RDD functions to be applied on a Dstream.
PairDStream Functions	updateStateByKey(function)	The updateStateByKey operation allows you to maintain arbitrary state while continuously updating it with new information.

RDD Persistence

Storage Level	Meaning
MEMORY_ONLY (default level)	Store RDD as deserialized Java objects. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly when needed.
MEMORY_AND_DISK	Store RDD as deserialized Java objects. If the RDD does not fit in memory, store the partitions that don't fit on disk, and load them when they're needed.
MEMORY_ONLY_SER	Store RDD as serialized Java objects. Generally more space-efficient than deserialized objects, but more CPU-intensive to read.
MEMORY_AND_DISK_SER	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc...	Same as the levels above, but replicate each partition on two cluster nodes.

Shared Data

Broadcast Variables Broadcast variables allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks.

Language	Create, Evaluate
Scala	val broadcastVar = sc.broadcast(Array(1, 2, 3)) broadcastVar.value
Java	Broadcast<int[]> broadcastVar = sc.broadcast(new int[] {1, 2, 3}); broadcastVar.value();
Python	broadcastVar = sc.broadcast([1, 2, 3]) broadcastVar.value

Accumulators Accumulators are variables that are only "added" to through an associative operation and can therefore be efficiently supported in parallel.

Language	Create, Add, Evaluate
Scala	val accum = sc.accumulator(0, My Accumulator) sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum += x) accum.value
Java	Accumulator<Integer> accum = sc.accumulator(0); sc.parallelize(Arrays.asList(1, 2, 3, 4)).foreach(x -> accum.add(x)) accum.value();
Python	accum = sc.accumulator(0)

MLlib Reference

Topic	Description
Data types	Vectors, points, matrices.
Basic Statistics	Summary, correlations, sampling, testing and random data.
Classification and regression	Includes SVMs, decision trees, naïve Bayes, etc...
Collaborative filtering	Commonly used for recommender systems.
Clustering	Clustering is an unsupervised learning approach.
Dimensionality reduction	Dimensionality reduction is the process of reducing the number of variables under consideration.
Feature extraction and transformation	Used in selecting a subset of relevant features (variables, predictors) for use in model construction.
Frequent pattern mining	Mining is usually among the first steps to analyze a large-scale dataset.
Optimization	Different optimization methods can have different convergence guarantees.
PMML model export	MLlib supports model export to Predictive Model Markup Language.

Other References

- **Launching Jobs**
- **SQL and DataFrames Programming Guide**
- **GraphX Programming Guide**
- **SparkR Programming Guide**

About the Author

James A. Scott (prefers to go by Jim) is Director, Enterprise Strategy & Architecture at MapR Technologies and is very active in the Hadoop community. Jim helped build the Hadoop community in Chicago as cofounder of the Chicago Hadoop Users Group. He has implemented Hadoop at three different companies, supporting a variety of enterprise use cases from managing Points of Interest for mapping applications, to Online Transactional Processing in advertising, as well as full data center monitoring and general data processing. Jim also was the SVP of Information Technology and Operations at SPINS, the leading provider of retail consumer insights, analytics reporting and consulting services for the Natural and Organic Products industry. Additionally, Jim served as Lead Engineer/Architect for Conversant (formerly Dotomi), one of the world's largest and most diversified digital marketing companies, and also held software architect positions at several companies including Aircell, NAVTEQ, and Dow Chemical. Jim speaks at many industry events around the world on big data technologies and enterprise architecture. When he's not solving business problems with technology, Jim enjoys cooking, watching-and-quoting movies and spending time with his wife and kids. Jim is on Twitter as ***@kingmesal****.*