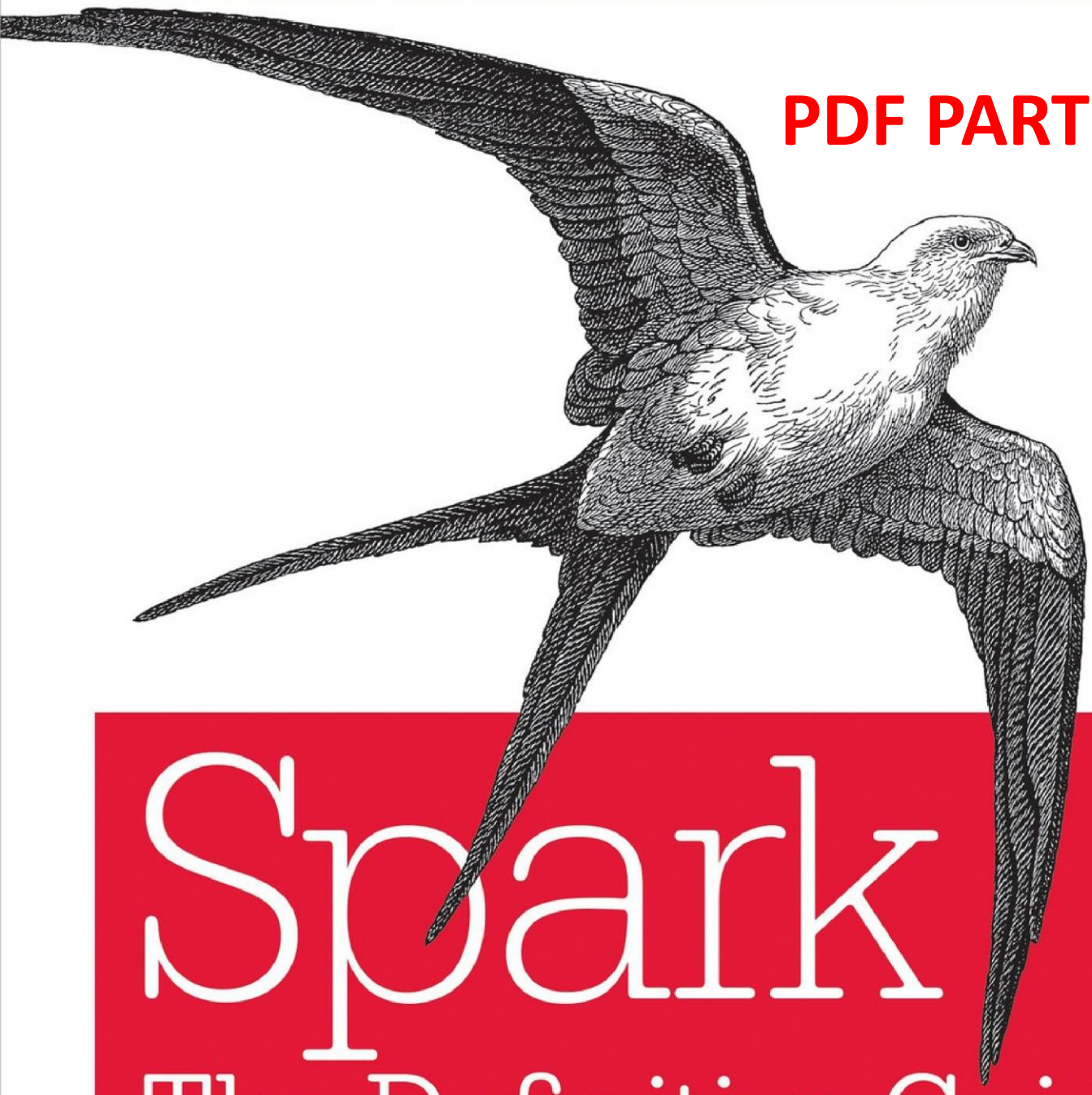
O'REILLY®

PDF PART 1



Spark
The Definitive Guide

BIG DATA PROCESSING MADE SIMPLE

Bill Chambers & Matei Zaharia

PDF PART 1

Chapt 1. What Is Apache Spark?

Apache Spark is a unified computing engine and a set of libraries for parallel data processing on computer clusters. As of this writing, Spark is the most actively developed open source engine for this task, making it a standard tool for any developer or data scientist interested in big data. Spark supports multiple widely used programming languages (Python, Java, Scala, and R), includes libraries for diverse tasks ranging from SQL to streaming and machine learning, and runs anywhere from a laptop to a cluster of thousands of servers. This makes it an easy system to start with and scale-up to big data processing or incredibly large scale.

Figure 1-1 illustrates all the components and libraries Spark offers to end-users.

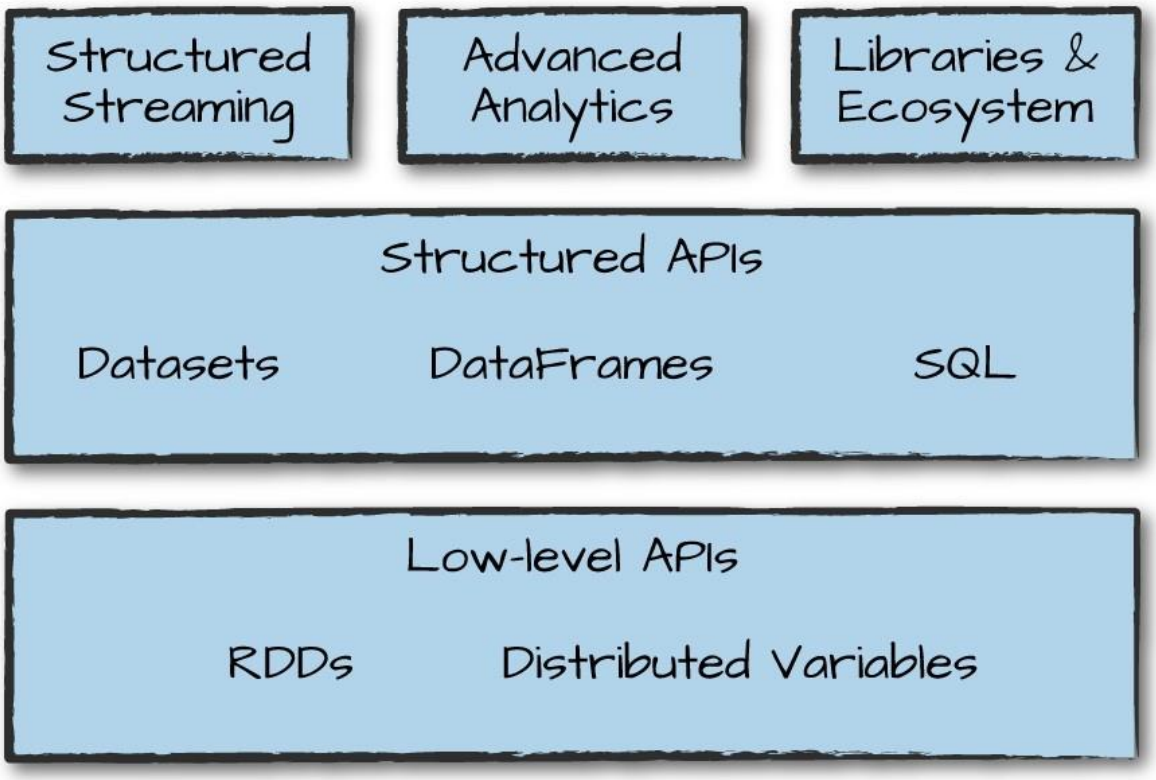


Figure 1-1. Spark's toolkit

You'll notice the categories roughly correspond to the different parts of this book. That should really come as no surprise; our goal here is to educate you on all aspects of Spark, and Spark is composed of a number of different components.

Given that you're reading this book, you might already know a little bit about Apache Spark and what it can do. Nonetheless, in this chapter, we want to briefly cover the overriding philosophy behind Spark as well as the context it was developed in (why is everyone suddenly excited about parallel data processing?) and its history. We will also outline the first few steps to running Spark.

Apache Spark's Philosophy

Let's break down our description of Apache Spark—a unified computing engine and set of libraries for big data—into its key components:

Unified

Spark's key driving goal is to offer a unified platform for writing big data applications. What do we mean by unified? Spark is designed to support a wide range of data analytics tasks, ranging from simple data loading and SQL queries to machine learning and streaming computation, over the same computing engine and with a consistent set of APIs. The main insight behind this goal is that real-world data analytics tasks—whether they are interactive analytics in a tool such as a Jupyter notebook, or traditional software development for production applications—tend to combine many different processing types and libraries.

Spark's unified nature makes these tasks both easier and more efficient to write. First, Spark provides consistent, composable APIs that you can use to build an application out of smaller pieces or out of existing libraries. It also makes it easy for you to write your own analytics libraries on top. However, composable APIs are not enough: Spark's APIs are also designed to enable high performance by optimizing across the different libraries and functions composed together in a user program. For example, if you load data using a SQL query and then evaluate a machine learning model over it using Spark's ML library, the engine can combine these steps into one scan over the data. The combination of general APIs and highperformance execution, no matter how you combine them, makes Spark a powerful platform for interactive and production applications.

Spark's focus on defining a unified platform is the same idea behind unified platforms in other areas of software. For example, data scientists benefit from a unified set of libraries (e.g., Python or R) when doing modeling, and web developers benefit from unified frameworks such as Node.js or Django. Before Spark, no open source systems tried to provide this type of unified engine for parallel data processing, meaning that users had to stitch together an application out of multiple APIs and systems. Thus, Spark quickly became the standard for this type of development. Over time, Spark has continued to expand its builtin APIs to cover more workloads. At the same time, the project's developers have continued to refine its theme of a unified engine. In particular, one major focus of this book

will be the “structured APIs” (DataFrames, Datasets, and SQL) that were finalized in Spark 2.0 to enable more powerful optimization under user applications.

Computing engine

At the same time that Spark strives for unification, it carefully limits its scope to a computing engine. By this, we mean that Spark handles loading data from storage systems and performing computation on it, not permanent storage as the end itself. You can use Spark with a wide variety of persistent storage systems, including cloud storage systems such as Azure Storage and Amazon S3, distributed file systems such as Apache Hadoop, key-value stores such as Apache Cassandra, and message buses such as Apache Kafka. However, Spark neither stores data long term itself, nor favors one over another. The key motivation here is that most data already resides in a mix of storage systems. Data is expensive to move so Spark focuses on performing computations over the data, no matter where it resides. In userfacing APIs, Spark works hard to make these storage systems look largely similar so that applications do not need to worry about where their data is.

Spark’s focus on computation makes it different from earlier big data software platforms such as Apache Hadoop. Hadoop included both a storage system (the Hadoop file system, designed for low-cost storage over clusters of commodity servers) and a computing system (MapReduce), which were closely integrated together. However, this choice makes it difficult to run one of the systems without the other. More important, this choice also makes it a challenge to write applications that access data stored anywhere else. Although Spark runs well on Hadoop storage, today it is also used broadly in environments for which the Hadoop architecture does not make sense, such as the public cloud (where storage can be purchased separately from computing) or streaming applications.

Libraries

Spark’s final component is its libraries, which build on its design as a unified engine to provide a unified API for common data analysis tasks. Spark supports both standard libraries that ship with the engine as well as a wide array of external libraries published as third-party packages by the open source communities. Today, Spark’s standard libraries are actually the bulk of the open source project: the Spark core engine itself has changed little since it was first released, but the libraries have grown to provide more and more types of functionality. Spark includes libraries for SQL and structured data (Spark SQL), machine learning (MLlib), stream processing (Spark Streaming and the newer Structured Streaming), and graph analytics (GraphX). Beyond these libraries, there are hundreds of open source external libraries ranging from connectors for various storage systems to machine learning algorithms. One index of external libraries is available at spark-packages.org.

Context: The Big Data Problem

Why do we need a new engine and programming model for data analytics in the first place? As with many trends in computing, this is due to changes in the economic factors that underlie computer applications and hardware.

For most of their history, computers became faster every year through processor speed increases: the new processors each year could run more instructions per second than the previous year's. As a result, applications also automatically became faster every year, without any changes needed to their code. This trend led to a large and established ecosystem of applications building up over time, most of which were designed to run only on a single processor. These applications rode the trend of improved processor speeds to scale up to larger computations and larger volumes of data over time.

Unfortunately, this trend in hardware stopped around 2005: due to hard limits in heat dissipation, hardware developers stopped making individual processors faster, and switched toward adding more parallel CPU cores all running at the same speed. This change meant that suddenly applications needed to be modified to add parallelism in order to run faster, which set the stage for new programming models such as Apache Spark.

On top of that, the technologies for storing and collecting data did not slow down appreciably in 2005, when processor speeds did. The cost to store 1 TB of data continues to drop by roughly two times every 14 months, meaning that it is very inexpensive for organizations of all sizes to store large amounts of data. Moreover, many of the technologies for collecting data (sensors, cameras, public datasets, etc.) continue to drop in cost and improve in resolution. For example, camera technology continues to improve in resolution and drop in cost per pixel every year, to the point where a 12-megapixel webcam costs only $3 to $4; this has made it inexpensive to collect a wide range of visual data, whether from people filming video or automated sensors in an industrial setting. Moreover, cameras are themselves the key sensors in other data collection devices, such as telescopes and even gene-sequencing machines, driving the cost of these technologies down as well.

The end result is a world in which collecting data is extremely inexpensive—many organizations today even consider it negligent *not* to log data of possible relevance to the business—but processing it requires large, parallel computations, often on clusters of machines. Moreover, in this new world, the software developed in the past 50 years cannot automatically scale up, and neither can the traditional programming models for data processing applications, creating the need for new programming models. It is this world that Apache Spark was built for.

History of Spark

Apache Spark began at UC Berkeley in 2009 as the Spark research project, which was first published the following year in a paper entitled "Spark: Cluster Computing with Working Sets"

by Matei Zaharia, Mosharaf Chowdhury, Michael Franklin, Scott Shenker, and Ion Stoica of the UC Berkeley AMPlab. At the time, Hadoop MapReduce was the dominant parallel programming engine for clusters, being the first open source system to tackle data-parallel processing on clusters of thousands of nodes. The AMPlab had worked with multiple early MapReduce users to understand the benefits and drawbacks of this new programming model, and was therefore able to synthesize a list of problems across several use cases and begin designing more general computing platforms. In addition, Zaharia had also worked with Hadoop users at UC Berkeley to understand their needs for the platform—specifically, teams that were doing large-scale machine learning using iterative algorithms that need to make multiple passes over the data.

Across these conversations, two things were clear. First, cluster computing held tremendous potential: at every organization that used MapReduce, brand new applications could be built using the existing data, and many new groups began using the system after its initial use cases. Second, however, the MapReduce engine made it both challenging and inefficient to build large applications. For example, the typical machine learning algorithm might need to make 10 or 20 passes over the data, and in MapReduce, each pass had to be written as a separate MapReduce job, which had to be launched separately on the cluster and load the data from scratch.

To address this problem, the Spark team first designed an API based on functional programming that could succinctly express multistep applications. The team then implemented this API over a new engine that could perform efficient, in-memory data sharing across computation steps. The team also began testing this system with both Berkeley and external users.

The first version of Spark supported only batch applications, but soon enough another compelling use case became clear: interactive data science and ad hoc queries. By simply plugging the Scala interpreter into Spark, the project could provide a highly usable interactive system for running queries on hundreds of machines. The AMPlab also quickly built on this idea to develop Shark, an engine that could run SQL queries over Spark and enable interactive use by analysts as well as data scientists. Shark was first released in 2011.

After these initial releases, it quickly became clear that the most powerful additions to Spark would be new libraries, and so the project began to follow the “standard library” approach it has today. In particular, different AMPlab groups started MLlib, Spark Streaming, and GraphX. They also ensured that these APIs would be highly interoperable, enabling writing end-to-end big data applications in the same engine for the first time.

In 2013, the project had grown to widespread use, with more than 100 contributors from more than 30 organizations outside UC Berkeley. The AMPlab contributed Spark to the Apache Software Foundation as a long-term, vendor-independent home for the project. The early AMPlab team also launched a company, Databricks, to harden the project, joining the community of other companies and organizations contributing to Spark. Since that time, the

Apache Spark community released Spark 1.0 in 2014 and Spark 2.0 in 2016, and continues to make regular releases, bringing new features into the project.

Finally, Spark's core idea of composable APIs has also been refined over time. Early versions of Spark (before 1.0) largely defined this API in terms of *functional operations*—parallel operations such as maps and reduces over collections of Java objects. Beginning with 1.0, the project added Spark SQL, a new API for working with *structured data*—tables with a fixed data format that is not tied to Java's in-memory representation. Spark SQL enabled powerful new optimizations across libraries and APIs by understanding both the data format and the user code that runs on it in more detail. Over time, the project added a plethora of new APIs that build on this more powerful structured foundation, including DataFrames, machine learning pipelines, and Structured Streaming, a high-level, automatically optimized streaming API. In this book, we will spend a signficant amount of time explaining these next-generation APIs, most of which are marked as production-ready.

The Present and Future of Spark

Spark has been around for a number of years but continues to gain in popularity and use cases.
Many new projects within the Spark ecosystem continue to push the boundaries of what's possible with the system. For example, a new high-level streaming engine, Structured Streaming, was introduced in 2016. This technology is a huge part of companies solving massive-scale data challenges, from technology companies like Uber and Netflix using Spark's streaming and machine learning tools, to institutions like NASA, CERN, and the Broad Institute of MIT and Harvard applying Spark to scientific data analysis.

Spark will continue to be a cornerstone of companies doing big data analysis for the foreseeable future, especially given that the project is still developing quickly. Any data scientist or engineer who needs to solve big data problems probably needs a copy of Spark on their machine—and hopefully, a copy of this book on their bookshelf!

Running Spark

This book contains an abundance of Spark-related code, and it's essential that you're prepared to run it as you learn. For the most part, you'll want to run the code interactively so that you can experiment with it. Let's go over some of your options before we begin working with the coding parts of the book.

You can use Spark from Python, Java, Scala, R, or SQL. Spark itself is written in Scala, and runs on the Java Virtual Machine (JVM), so therefore to run Spark either on your laptop or a cluster, all you need is an installation of Java. If you want to use the Python API, you will also need a Python interpreter (version 2.7 or later). If you want to use R, you will need a version of R on your machine.

There are two options we recommend for getting started with Spark: downloading and installing Apache Spark on your laptop, or running a web-based version in Databricks Community Edition, a free cloud environment for learning Spark that includes the code in this book. We explain both of those options next.

Downloading Spark Locally

If you want to download and run Spark locally, the first step is to make sure that you have Java installed on your machine (available as j ava), as well as a Python version if you would like to use Python. Next, visit the project's official download page, select the package type of "Pre-built for Hadoop 2.7 and later," and click "Direct Download." This downloads a compressed TAR file, or tarball, that you will then need to extract. The majority of this book was written using Spark 2.2, so downloading version 2.2 or later should be a good starting point.

Downloading Spark for a Hadoop cluster

Spark can run locally without any distributed storage system, such as Apache Hadoop. However, if you would like to connect the Spark version on your laptop to a Hadoop cluster, make sure you download the right Spark version for that Hadoop version, which can be chosen at *http://spark.apache.org/downloads.html* by selecting a different package type. We discuss how
Spark runs on clusters and the Hadoop file system in later chapters, but at this point we recommend just running Spark on your laptop to start out.

> **NOTE**
>
> In Spark 2.2, the developers also added the ability to install Spark for Python via pi p i nst al l pyspark. This functionality came out as this book was being written, so we weren't able to include all of the relevant instructions.

Building Spark from source

We won't cover this in the book, but you can also build and configure Spark from source. You can select a source package on the Apache download page to get just the source and follow the instructions in the README file for building.

After you've downloaded Spark, you'll want to open a command-line prompt and extract the package. In our case, we're installing Spark 2.2. The following is a code snippet that you can run on any Unix-style command line to unzip the file you downloaded from Spark and move into the directory:

```
cd ~/Downl oads t ar - xf spark- 2. 2. 0- bi n-
hadoop2. 7. t gz cd spark- 2. 2. 0- bi n-
hadoop2. 7. t gz
```

Note that Spark has a large number of directories and files within the project. Don't be intimidated! Most of these directories are relevant only if you're reading source code. The next section will cover the most important directories—the ones that let us launch Spark's different consoles for interactive use.

Launching Spark's Interactive Consoles

You can start an interactive shell in Spark for several different programming languages. The majority of this book is written with Python, Scala, and SQL in mind; thus, those are our recommended starting points.

Launching the Python console

You'll need Python 2 or 3 installed in order to launch the Python console. From Spark's home directory, run the following code:

```
. /bi n/pyspark
```

After you've done that, type "spark" and press Enter. You'll see the SparkSession object printed, which we cover in Chapter 2.

Launching the Scala console

To launch the Scala console, you will need to run the following command:

```
. /bi n/spark- shel l
```

After you've done that, type "spark" and press Enter. As in Python, you'll see the SparkSession object, which we cover in Chapter 2.

Launching the SQL console

Parts of this book will cover a large amount of Spark SQL. For those, you might want to start the SQL console. We'll revisit some of the more relevant details after we actually cover these topics in the book.

```
. /bi n/spark- sql
```

Running Spark in the Cloud

If you would like to have a simple, interactive notebook experience for learning Spark, you might prefer using Databricks Community Edition. Databricks, as we mentioned earlier, is a company founded by the Berkeley team that started Spark, and offers a free community edition of its cloud service as a learning environment. The Databricks Community Edition includes a copy of all the data and code examples for this book, making it easy to quickly run

any of them. To use the Databricks Community Edition, follow the instructions at *https://github.com/databricks/Spark-The-Definitive-Guide*. You will be able to use Scala, Python, SQL, or R from a web browser–based interface to run and visualize results.

Data Used in This Book

We'll use a number of data sources in this book for our examples. If you want to run the code locally, you can download them from the official code repository in this book as desribed at *https://github.com/databricks/Spark-The-Definitive-Guide*. In short, you will download the data, put it in a folder, and then run the code snippets in this book!

Chapter 2. A Gentle Introduction to Spark

Now that our history lesson on Apache Spark is completed, it's time to begin using and applying it! This chapter presents a gentle introduction to Spark, in which we will walk through the core architecture of a cluster, Spark Application, and Spark's structured APIs using DataFrames and SQL. Along the way we will touch on Spark's core terminology and concepts so that you can begin using Spark right away. Let's get started with some basic background information.

Spark's Basic Architecture

Typically, when you think of a "computer," you think about one machine sitting on your desk at home or at work. This machine works perfectly well for watching movies or working with spreadsheet software. However, as many users likely experience at some point, there are some things that your computer is not powerful enough to perform. One particularly challenging area is data processing. Single machines do not have enough power and resources to perform computations on huge amounts of information (or the user probably does not have the time to wait for the computation to finish). A *cluster*, or group, of computers, pools the resources of many machines together, giving us the ability to use all the cumulative resources as if they were a single computer. Now, a group of machines alone is not powerful, you need a framework to coordinate work across them. Spark does just that, managing and coordinating the execution of tasks on data across a cluster of computers.

The cluster of machines that Spark will use to execute tasks is managed by a cluster manager like Spark's standalone cluster manager, YARN, or Mesos. We then submit Spark Applications to these cluster managers, which will grant resources to our application so that we can complete our work.

Spark Applications

Spark Applications consist of a *driver* process and a set of *executor* processes. The driver process runs your mai n( ) function, sits on a node in the cluster, and is responsible for three things: maintaining information about the Spark Application; responding to a user's program or input; and analyzing, distributing, and scheduling work across the executors (discussed momentarily). The driver process is absolutely essential—it's the heart of a Spark Application and maintains all relevant information during the lifetime of the application.

The *executors* are responsible for actually carrying out the work that the driver assigns them. This means that each executor is responsible for only two things: executing code assigned to it by the driver, and reporting the state of the computation on that executor back to the driver node.

Figure 2-1 demonstrates how the cluster manager controls physical machines and allocates resources to Spark Applications. This can be one of three core cluster managers: Spark's standalone cluster manager, YARN, or Mesos. This means that there can be multiple Spark Applications running on a cluster at the same time. We will discuss cluster managers more in Part IV.

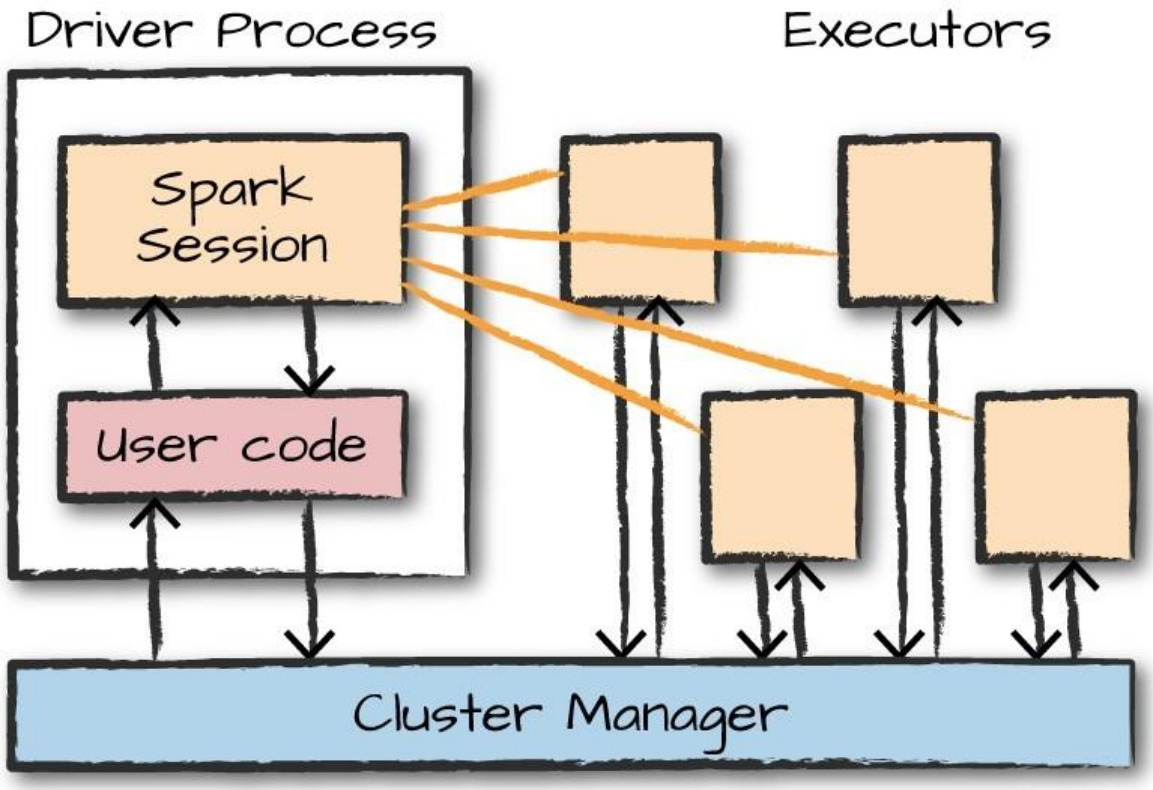


Figure 2-1. The architecture of a Spark Application

In Figure 2-1, we can see the driver on the left and four executors on the right. In this diagram, we removed the concept of cluster nodes. The user can specify how many executors should fall on each node through configurations.

> **NOTE**
>
> Spark, in addition to its cluster mode, also has a *local mode*. The driver and executors are simply processes, which means that they can live on the same machine or different machines. In local mode, the driver and executurs run (as threads) on your individual computer instead of a cluster. We wrote this book with local mode in mind, so you should be able to run everything on a single machine.

Here are the key points to understand about Spark Applications at this point:

- Spark employs a cluster manager that keeps track of the resources available.
- The driver process is responsible for executing the driver program's commands across the executors to complete a given task.

The executors, for the most part, will always be running Spark code. However, the driver can be "driven" from a number of different languages through Spark's language APIs. Let's take a look at those in the next section.

Spark's Language APIs

Spark's language APIs make it possible for you to run Spark code using various programming languages. For the most part, Spark presents some core "concepts" in every language; these concepts are then translated into Spark code that runs on the cluster of machines. If you use just the Structured APIs, you can expect all languages to have similar performance characteristics. Here's a brief rundown:

Scala

Spark is primarily written in Scala, making it Spark's "default" language. This book will include Scala code examples wherever relevant.

Java

Even though Spark is written in Scala, Spark's authors have been careful to ensure that you can write Spark code in Java. This book will focus primarily on Scala but will provide Java examples where relevant.

Python

Python supports nearly all constructs that Scala supports. This book will include Python code examples whenever we include Scala code examples and a Python API exists.

SQL

Spark supports a subset of the ANSI SQL 2003 standard. This makes it easy for analysts and non-programmers to take advantage of the big data powers of Spark. This book includes SQL code examples wherever relevant.

R

Spark has two commonly used R libraries: one as a part of Spark core (SparkR) and another as an R community-driven package (sparklyr). We cover both of these integrations in Chapter 32.

Figure 2-2 presents a simple illustration of this relationship.

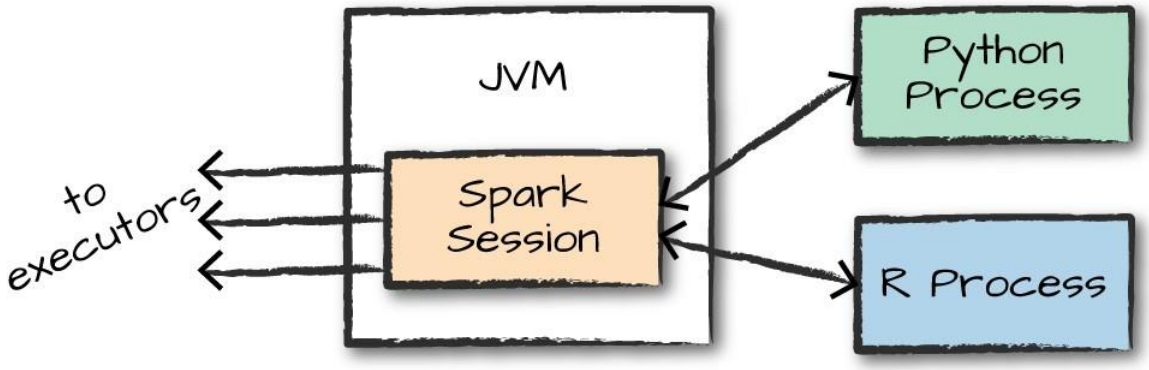


Figure 2-2. The relationship between the SparkSession and Spark's Language API

Each language API maintains the same core concepts that we described earlier. There is a SparkSession object available to the user, which is the entrance point to running Spark code. When using Spark from Python or R, you don't write explicit JVM instructions; instead, you write Python and R code that Spark translates into code that it then can run on the executor JVMs.

Spark's APIs

Although you can drive Spark from a variety of languages, what it makes available in those languages is worth mentioning. Spark has two fundamental sets of APIs: the low-level "unstructured" APIs, and the higher-level structured APIs. We discuss both in this book, but these introductory chapters will focus primarily on the higher-level structured APIs.

Starting Spark

Thus far, we covered the basic concepts of Spark Applications. This has all been conceptual in nature. When we actually go about writing our Spark Application, we are going to need a way to send user commands and data to it. We do that by first creating a SparkSession.

> **NOTE**
>
> To do this, we will start Spark's local mode, just like we did in Chapter 1. This means running . /bi n/spark- shel l to access the Scala console to start an interactive session. You can also start the Python console by using . /bi n/pyspark. This starts an interactive Spark Application. There is also a process for submitting standalone applications to Spark called spark- submi t , whereby you can submit a precompiled application to Spark. We'll show you how to do that in Chapter 3.

When you start Spark in this interactive mode, you implicitly create a SparkSession that manages the Spark Application. When you start it through a standalone application, you must create the SparkSession object yourself in your application code.

The SparkSession

As discussed in the beginning of this chapter, you control your Spark Application through a driver process called the SparkSession. The SparkSession instance is the way Spark executes user-defined manipulations across the cluster. There is a one-to-one correspondence between a SparkSession and a Spark Application. In Scala and Python, the variable is available as spark when you start the console. Let's go ahead and look at the SparkSessi on in both Scala and/or Python: spark

In Scala, you should see something like the following: res0: org. apache. spark. sql .

```
SparkSessi on = org. apache. spark. sql . SparkSessi on@. . .
```

In Python you'll see something like this:

```
<pyspark. sql . sessi on. SparkSessi on at 0x7ef da4c1ccd0>
```

Let's now perform the simple task of creating a range of numbers. This range of numbers is just like a named column in a spreadsheet:

```
// i n Scal a val myRange = spark. range( 1000) . t oDF(
"number")

# i n Pyt hon
myRange = spark. range( 1000) . t oDF( "number")
```

You just ran your first Spark code! We created a *DataFrame* with one column containing 1,000 rows with values from 0 to 999. This range of numbers represents a *distributed collection*. When run on a cluster, each part of this range of numbers exists on a different executor. This is a Spark DataFrame.

DataFrames

A DataFrame is the most common Structured API and simply represents a table of data with rows and columns. The list that defines the columns and the types within those columns is called the *schema*. You can think of a DataFrame as a spreadsheet with named columns. Figure 2-3 illustrates the fundamental difference: a spreadsheet sits on one computer in one specific location, whereas a Spark DataFrame can span thousands of computers. The reason for putting the data on more than one computer should be intuitive: either the data is too large to fit on one machine or it would simply take too long to perform that computation on one machine.

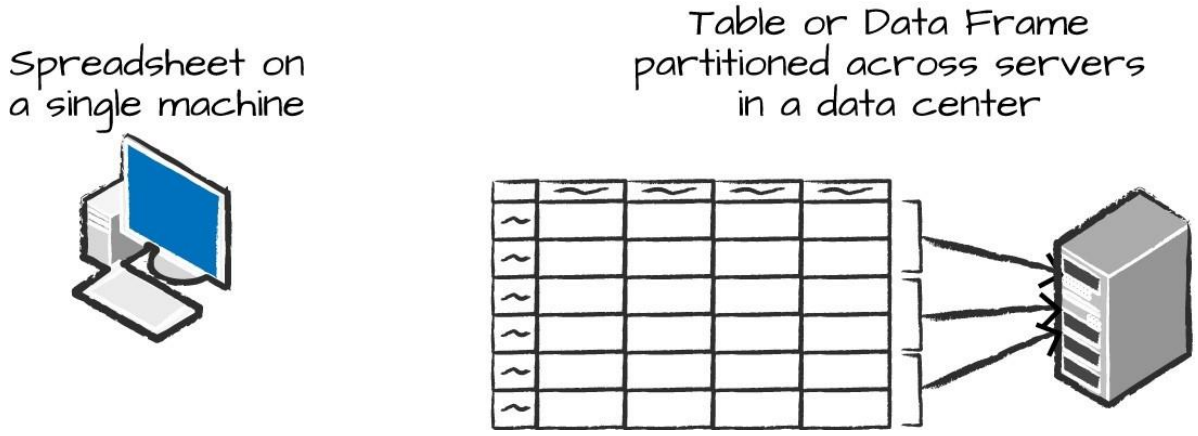


Figure 2-3. Distributed versus single-machine analysis

The DataFrame concept is not unique to Spark. R and Python both have similar concepts. However, Python/R DataFrames (with some exceptions) exist on one machine rather than multiple machines. This limits what you can do with a given DataFrame to the resources that exist on that specific machine. However, because Spark has language interfaces for both Python and R, it's quite easy to convert Pandas (Python) DataFrames to Spark DataFrames, and R DataFrames to Spark DataFrames.

> **NOTE**
>
> Spark has several core abstractions: Datasets, DataFrames, SQL Tables, and Resilient Distributed Datasets (RDDs). These different abstractions all represent distributed collections of data. The easiest and most efficient are DataFrames, which are available in all languages. We cover Datasets at the end of Part II, and RDDs in Part III.

Partitions

To allow every executor to perform work in parallel, Spark breaks up the data into chunks called *partitions*. A partition is a collection of rows that sit on one physical machine in your cluster. A DataFrame's partitions represent how the data is physically distributed across the

cluster of machines during execution. If you have one partition, Spark will have a parallelism of only one, even if you have thousands of executors. If you have many partitions but only one executor, Spark will still have a parallelism of only one because there is only one computation resource.

An important thing to note is that with DataFrames you do not (for the most part) manipulate partitions manually or individually. You simply specify high-level transformations of data in the physical partitions, and Spark determines how this work will actually execute on the cluster. Lower-level APIs do exist (via the RDD interface), and we cover those in Part III.

Transformations

In Spark, the core data structures are *immutable*, meaning they cannot be changed after they're created. This might seem like a strange concept at first: if you cannot change it, how are you supposed to use it? To "change" a DataFrame, you need to instruct Spark how you would like to modify it to do what you want. These instructions are called *transformations*. Let's perform a simple transformation to find all even numbers in our current DataFrame:

```
// i n Scal a val di vi sBy2 = myRange. where( "number % 2
= 0")

# i n Pyt hon
di vi sBy2 = myRange. where( "number % 2 = 0")
```

Notice that these return no output. This is because we specified only an abstract transformation, and Spark will not act on transformations until we call an action (we discuss this shortly). Transformations are the core of how you express your business logic using Spark. There are two types of transformations: those that specify *narrow dependencies*, and those that specify *wide dependencies*.

Transformations consisting of narrow dependencies (we'll call them narrow transformations) are those for which each input partition will contribute to only one output partition. In the preceding code snippet, the where statement specifies a narrow dependency, where only one partition contributes to at most one output partition, as you can see in Figure 2-4.

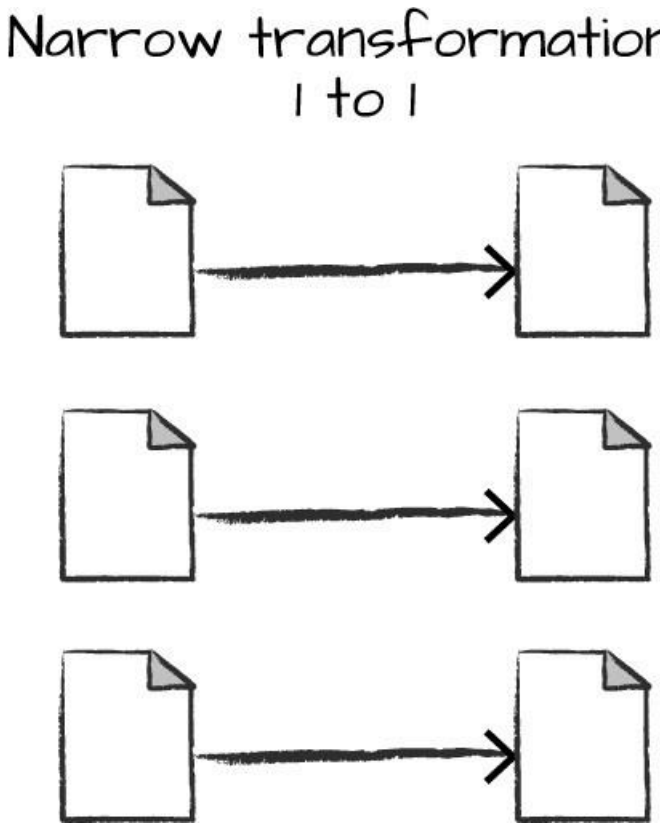


Figure 2-4. A narrow dependency

A wide dependency (or wide transformation) style transformation will have input partitions contributing to many output partitions. You will often hear this referred to as a *shuffle* whereby Spark will exchange partitions across the cluster. With narrow transformations, Spark will automatically perform an operation called *pipelining*, meaning that if we specify multiple filters on DataFrames, they'll all be performed in-memory. The same cannot be said for shuffles. When we perform a shuffle, Spark writes the results to disk. Wide transformations are illustrated in Figure 2-5.

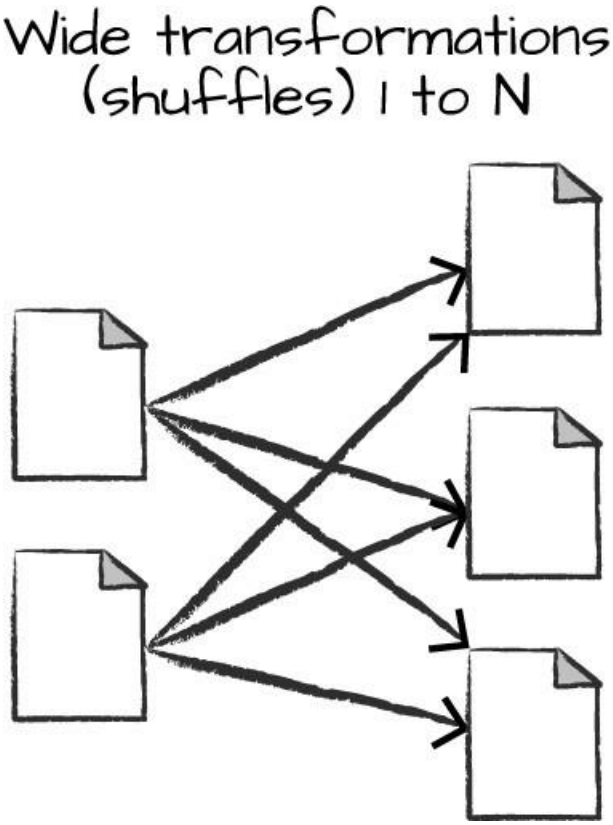


Figure 2-5. A wide dependency

You'll see a lot of discussion about shuffle optimization across the web because it's an important topic, but for now, all you need to understand is that there are two kinds of transformations. You now can see how transformations are simply ways of specifying different series of data manipulation. This leads us to a topic called *lazy evaluation*.

Lazy Evaluation

Lazy evaulation means that Spark will wait until the very last moment to execute the graph of computation instructions. In Spark, instead of modifying the data immediately when you express some operation, you build up a *plan* of transformations that you would like to apply to your source data. By waiting until the last minute to execute the code, Spark compiles this plan from your raw DataFrame transformations to a streamlined physical plan that will run as efficiently as possible across the cluster. This provides immense benefits because Spark can optimize the entire data flow from end to end. An example of this is something called *predicate pushdown* on DataFrames. If we build a large Spark job but specify a filter at the end that only requires us to fetch one row from our source data, the most efficient way to execute this is to access the single record that we need. Spark will actually optimize this for us by pushing the filter down automatically.

Actions

Transformations allow us to build up our logical transformation plan. To trigger the computation, we run an *action*. An action instructs Spark to compute a result from a series of transformations.

The simplest action is count , which gives us the total number of records in the DataFrame:

```
divisBy2.count()
```

The output of the preceding code should be 500. Of course, count is not the only action. There are three kinds of actions:

- Actions to view data in the console
- Actions to collect data to native objects in the respective language
- Actions to write to output data sources

In specifying this action, we started a Spark job that runs our filter transformation (a narrow transformation), then an aggregation (a wide transformation) that performs the counts on a per partition basis, and then a collect, which brings our result to a native object in the respective language. You can see all of this by inspecting the Spark UI, a tool included in Spark with which you can monitor the Spark jobs running on a cluster.

Spark UI

You can monitor the progress of a job through the Spark web UI. The Spark UI is available on port 4040 of the driver node. If you are running in local mode, this will be *http://localhost:4040*. The Spark UI displays information on the state of your Spark jobs, its environment, and cluster state. It's very useful, especially for tuning and debugging. Figure 2-6 shows an example UI for a Spark job where two stages containing nine tasks were executed.

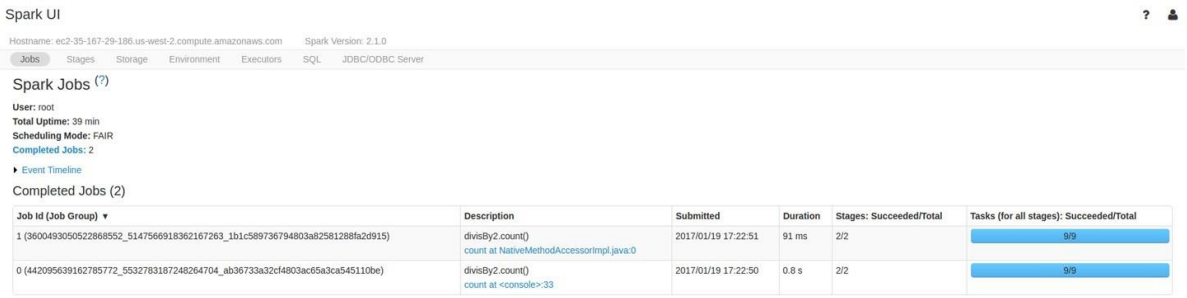


Figure 2-6. The Spark UI

This chapter will not go into detail about Spark job execution and the Spark UI. We will cover that in Chapter 18. At this point, all you need to understand is that a Spark *job* represents a set of transformations triggered by an individual action, and you can monitor that job from the Spark UI.

An End-to-End Example

In the previous example, we created a DataFrame of a range of numbers; not exactly groundbreaking big data. In this section, we will reinforce everything we learned previously in this chapter with a more realistic example, and explain step by step what is happening under the hood. We'll use Spark to analyze some flight data from the United States Bureau of Transportation statistics.

Inside of the CSV folder, you'll see that we have a number of files. There's also a number of other folders with different file formats, which we discuss in Chapter 9. For now, let's focus on the CSV files.

Each file has a number of rows within it. These files are CSV files, meaning that they're a semistructured data format, with each row in the file representing a row in our future DataFrame:

```
$ head /dat a/f l i ght - dat a/csv/2015- summary. csv

DEST_COUNTRY_NAME, ORI GI N_COUNTRY_NAME, count
Uni t ed St at es, Romani a, 15
Uni t ed St at es, Croat i a, 1
Uni t ed St at es, I rel and, 344
```

Spark includes the ability to read and write from a large number of data sources. To read this data, we will use a DataFrameReader that is associated with our SparkSession. In doing so, we will specify the file format as well as any options we want to specify. In our case, we want to do something called *schema inference*, which means that we want Spark to take a best guess at what the schema of our DataFrame should be. We also want to specify that the first row is the header in the file, so we'll specify that as an option, too.

To get the schema information, Spark reads in a little bit of the data and then attempts to parse the types in those rows according to the types available in Spark. You also have the option of strictly specifying a schema when you read in data (which we recommend in production scenarios):

```
// in Scala val flightData2015
= spark
  .read
  .option("inferSchema", "true")
  .option("header", "true")
  .csv("/data/flight-data/csv/2015-summary.csv")

# in Python
flightData2015 = spark\
  .read\
  .option("inferSchema", "true") \
  .option("header", "true") \
  .csv("/data/flight-data/csv/2015-summary.csv")
```

Each of these DataFrames (in Scala and Python) have a set of columns with an unspecified number of rows. The reason the number of rows is unspecified is because reading data is a transformation, and is therefore a lazy operation. Spark peeked at only a couple of rows of data to try to guess what types each column should be. Figure 2-7 provides an illustration of the CSV file being read into a DataFrame and then being converted into a local array or list of rows.

Figure 2-7. Reading a CSV file into a DataFrame and converting it to a local array or list of rows

If we perform the take action on the DataFrame, we will be able to see the same results that we saw before when we used the command line: `flightData2015.take(3)`

```
Array([United States,Romania,15], [United States,Croatia...
```

Let's specify some more transformations! Now, let's sort our data according to the count column, which is an integer type. Figure 2-8 illustrates this process.

> **NOTE**
>
> Remember, sort does not modify the DataFrame. We use sort as a transformation that returns a new DataFrame by transforming the previous DataFrame. Let's illustrate what's happening when we call t ake on that resulting DataFrame (Figure 2-8).

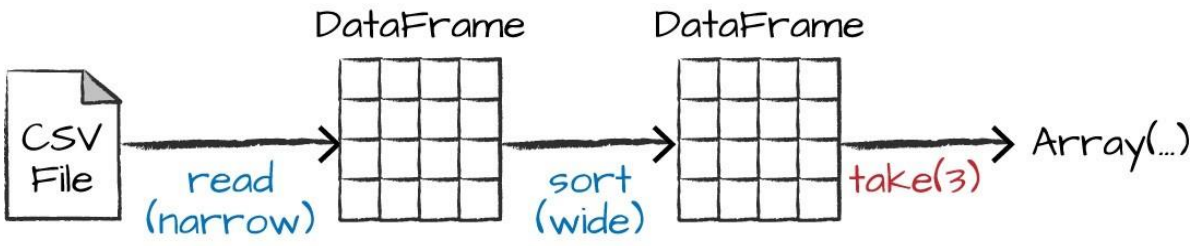


Figure 2-8. Reading, sorting, and collecting a DataFrame

Nothing happens to the data when we call sort because it's just a transformation. However, we can see that Spark is building up a plan for how it will execute this across the cluster by looking at the expl ai n plan. We can call expl ai n on any DataFrame object to see the DataFrame's lineage (or how Spark will execute this query): f l i ght Dat a2015. sort ("count ") . expl ai n( )

```
== Physi cal Pl an ==
*Sort [ count #195 ASC NULLS FI RST] , t rue, 0
+- Exchange rangepart i t i oni ng( count #195 ASC NULLS FI RST, 200)
  +- *Fi l eScan csv [ DEST_COUNTRY_NAME#193, ORI GI N_COUNTRY_NAME#194, count #195] . . .
```

Congratulations, you've just read your first explain plan! Explain plans are a bit arcane, but with a bit of practice it becomes second nature. You can read explain plans from top to bottom, the top being the end result, and the bottom being the source(s) of data. In this case, take a look at the first keywords. You will see sort, exchange, and FileScan. That's because the sort of our data is actually a wide transformation because rows will need to be compared with one another. Don't worry too much about understanding everything about explain plans at this point, they can just be helpful tools for debugging and improving your knowledge as you progress with Spark.

Now, just like we did before, we can specify an action to kick off this plan. However, before doing that, we're going to set a configuration. By default, when we perform a shuffle, Spark outputs 200 shuffle partitions. Let's set this value to 5 to reduce the number of the output partitions from the shuffle:

```
spark. conf . set ( "spark. sql . shuf f l e. part i t i ons", "5")
```

```
f l i ght Dat a2015. sort ( "count ") . t ake( 2)

. . . Array( [ Uni t ed St at es, Si ngapore, 1] , [ Mol dova, Uni t ed St at es, 1] )
```

Figure 2-9 illustrates this operation. Notice that in addition to the logical transformations, we include the physical partition count, as well.

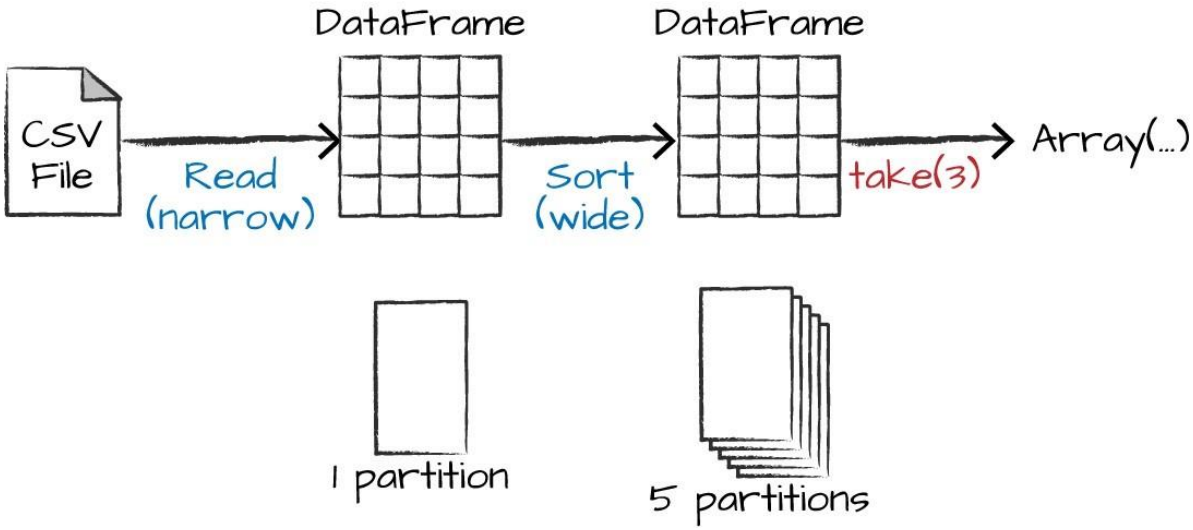


Figure 2-9. The process of logical and physical DataFrame manipulation

The logical plan of transformations that we build up defines a lineage for the DataFrame so that at any given point in time, Spark knows how to recompute any partition by performing all of the operations it had before on the same input data. This sits at the heart of Spark's programming model—functional programming where the same inputs always result in the same outputs when the transformations on that data stay constant.

We do not manipulate the physical data; instead, we configure physical execution characteristics through things like the shuffle partitions parameter that we set a few moments ago. We ended up with five output partitions because that's the value we specified in the shuffle partition. You can change this to help control the physical execution characteristics of your Spark jobs. Go ahead and experiment with different values and see the number of partitions yourself. In experimenting with different values, you should see drastically different runtimes. Remember that you can monitor the job progress by navigating to the Spark UI on port 4040 to see the physical and logical execution characteristics of your jobs.

DataFrames and SQL

We worked through a simple transformation in the previous example, let's now work through a more complex one and follow along in both DataFrames and SQL. Spark can run the same transformations, regardless of the language, in the exact same way. You can express your business logic in SQL or DataFrames (either in R, Python, Scala, or Java) and Spark will compile that logic down to an underlying plan (that you can see in the explain plan) before actually executing your code. With Spark SQL, you can register any DataFrame as a table or view (a

temporary table) and query it using pure SQL. There is no performance difference between writing SQL queries or writing DataFrame code, they both “compile” to the same underlying plan that we specify in DataFrame code.

You can make any DataFrame into a table or view with one simple method call:

```
f l i ght Dat a2015. creat eOrRepl aceTempVi ew( "f l i ght _dat a_2015")
```

Now we can query our data in SQL. To do so, we’ll use the spark. sql function (remember, spark is our SparkSession variable) that conveniently returns a new DataFrame. Although this might seem a bit circular in logic—that a SQL query against a DataFrame returns another DataFrame—it’s actually quite powerful. This makes it possible for you to specify transformations in the manner most convenient to you at any given point in time and not sacrifice any efficiency to do so! To understand that this is happening, let’s take a look at two explain plans:

```
// i n Scal a
val sql Way = spark. sql ( """
SELECT DEST_COUNTRY_NAME, count ( 1)
FROM f l i ght _dat a_2015
GROUP BY DEST_COUNTRY_NAME
""")

val dat aFrameWay = f l i ght Dat a2015
  . groupBy( ' DEST_COUNTRY_NAME)
  . count ( )

sql  Way.  expl  ai  n  dat
aFrameWay. expl ai n

# i n Pyt hon
sql Way = spark. sql ( """
SELECT DEST_COUNTRY_NAME, count ( 1)
FROM f l i ght _dat a_2015
GROUP BY DEST_COUNTRY_NAME
""")

dat aFrameWay = f l i ght Dat a2015\
  . groupBy( "DEST_COUNTRY_NAME") \
  . count ( )

sql  Way.  expl  ai  n(  )  dat
aFrameWay. expl ai n( )

== Physi cal Pl an ==
*HashAggregat e( keys=[ DEST_COUNTRY_NAME#182] , f unct i ons=[ count ( 1) ] )
+- Exchange hashpart i t i oni ng( DEST_COUNTRY_NAME#182, 5)
```

```
   +- *HashAggregate( keys=[DEST_COUNTRY_NAME#182], functions=[partial_count(1)])      +-
*FileScan csv [DEST_COUNTRY_NAME#182] ...
== Physical Plan ==
*HashAggregate( keys=[DEST_COUNTRY_NAME#182], functions=[count(1)])
+- Exchange hashpartitioning( DEST_COUNTRY_NAME#182, 5)
   +- *HashAggregate( keys=[DEST_COUNTRY_NAME#182], functions=[partial_count(1)])      +-
*FileScan csv [DEST_COUNTRY_NAME#182] ...
```

Notice that these plans compile to the exact same underlying plan!

Let's pull out some interesting statistics from our data. One thing to understand is that DataFrames (and SQL) in Spark already have a huge number of manipulations available. There are hundreds of functions that you can use and import to help you resolve your big data problems faster. We will use the max function, to establish the maximum number of flights to and from any given location. This just scans each value in the relevant column in the DataFrame and checks whether it's greater than the previous values that have been seen. This is a transformation, because we are effectively filtering down to one row. Let's see what that looks like: `spark.sql("SELECT max(count) from flight_data_2015").take(1)`

```
// in Scala import org.apache.spark.sql.functions.
maxflight Data2015.select(max("count")).take(
1)

# in Python from pyspark.sql.functions import
maxflight Data2015.select(max("count")).take(
1)
```

Great, that's a simple example that gives a result of 370,002. Let's perform something a bit more complicated and find the top five destination countries in the data. This is our first multitransformation query, so we'll take it step by step. Let's begin with a fairly straightforward SQL aggregation:

```
// in Scala
val maxSql = spark.sql("""
SELECT DEST_COUNTRY_NAME, sum(count) as destination_total
FROM flight_data_2015
GROUP BY DEST_COUNTRY_NAME
ORDER BY sum(count) DESC
LIMIT 5""")
maxSql.show(
)

# in Python
maxSql = spark.sql("""
```

```
SELECT DEST_COUNTRY_NAME, sum( count ) as dest i nat i on_t ot al
FROM f l i ght _dat a_2015
GROUP BY DEST_COUNTRY_NAME
ORDER BY sum( count ) DESC
LI MI T 5 """)
maxSql . show(
) +- - - - - - - - - -
- - - - - - - +- - - -
- - - - - - - - - - - -
- +
|DEST_COUNT
RY_NAME|dest
i nat i on_t ot al
|
+- - - - - - - - - - - - - - - - +- - - - - - - - - - - - - - - - +
|   Uni t ed St at es|        411352|
|        Canada|          8399|
|        Mexi co|          7140|
|  Uni t ed Ki ngdom|          2025|
|        J apan|          1548|
+- - - - - - - - - - - - - - - - +- - - - - - - - - - - - - - - - +
```

Now, let's move to the DataFrame syntax that is semantically similar but slightly different in implementation and ordering. But, as we mentioned, the underlying plans for both of them are the same. Let's run the queries and see their results as a sanity check:

```
// i n Scal a i mport org. apache. spark. sql . f unct i
ons. desc

f l i ght Dat a2015
  . groupBy( "DEST_COUNTRY_NAME")
  . sum( "count ")
  . wi t hCol umnRenamed( "sum( count ) ", "dest i nat i on_t ot al ")
  . sort ( desc( "dest i nat i on_t ot al ") )
  . l i mi t ( 5)
  . show( )

# i n Pyt hon f rom pyspark. sql . f unct i ons i
mport desc

f l i ght Dat a2015\
  . groupBy( "DEST_COUNTRY_NAME") \
  . sum( "count ") \
  . wi t hCol umnRenamed( "sum( count ) ", "dest i nat i on_t ot al ") \
```

```
  . sort ( desc( "dest i nat i on_t ot al ") ) \
  . l i mi t ( 5) \
  . show( )

+-----------------+-----------------+ |DEST_COUNTRY_NAME|dest
i nat i on_t ot al |
+-----------------+-----------------+
|   Uni t ed St at es|         411352|
|         Canada|           8399|
|         Mexi co|           7140|
|   Uni t ed Ki ngdom|           2025|
|         J apan|           1548|
+-----------------+-----------------+
```

Now there are seven steps that take us all the way back to the source data. You can see this in the explain plan on those DataFrames. Figure 2-10 shows the set of steps that we perform in "code."

The true execution plan (the one visible in expl ai n) will differ from that shown in Figure 2-10 because of optimizations in the physical execution; however, the llustration is as good of a starting point as any. This execution plan is a *directed acyclic graph* (DAG) of transformations, each resulting in a new immutable DataFrame, on which we call an action to generate a result.

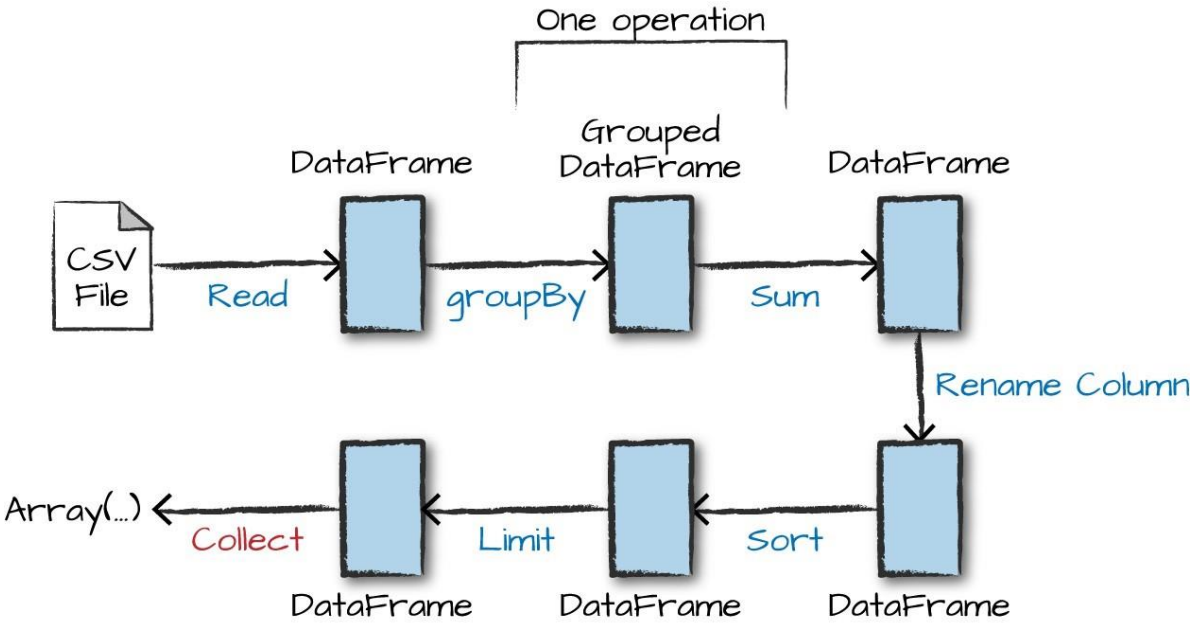


Figure 2-10. The entire DataFrame transformation flow

The first step is to read in the data. We defined the DataFrame previously but, as a reminder, Spark does not actually read it in until an action is called on that DataFrame or one derived from the original DataFrame.

The second step is our grouping; technically when we call groupBy, we end up with a

Rel at i onal GroupedDat aset , which is a fancy name for a DataFrame that has a grouping specified but needs the user to specify an aggregation before it can be queried further. We basically specified that we're going to be grouping by a key (or set of keys) and that now we're going to perform an aggregation over each one of those keys.

Therefore, the third step is to specify the aggregation. Let's use the sum aggregation method. This takes as input a column expression or, simply, a column name. The result of the sum method call is a new DataFrame. You'll see that it has a new schema but that it does know the type of each column. It's important to reinforce (again!) that no computation has been performed. This is simply another transformation that we've expressed, and Spark is simply able to trace our type information through it.

The fourth step is a simple renaming. We use the wi t hCol umnRenamed method that takes two arguments, the original column name and the new column name. Of course, this doesn't perform computation: this is just another transformation!

The fifth step sorts the data such that if we were to take results off of the top of the DataFrame, they would have the largest values in the dest i nat i on_t ot al column.

You likely noticed that we had to import a function to do this, the desc function. You might also have noticed that desc does not return a string but a Col umn. In general, many DataFrame methods will accept strings (as column names) or Col umn types or expressions. Columns and expressions are actually the exact same thing.

Penultimately, we'll specify a limit. This just specifies that we only want to return the first five values in our final DataFrame instead of all the data.

The last step is our action! Now we actually begin the process of collecting the results of our DataFrame, and Spark will give us back a list or array in the language that we're executing. To reinforce all of this, let's look at the explain plan for the previous query:

```
// i n Scal a f l i ght
Dat a2015
  . groupBy( "DEST_COUNTRY_NAME")
  . sum( "count ")
  . wi t hCol umnRenamed( "sum( count ) ", "dest i nat i on_t ot al ")
  . sort ( desc( "dest i nat i on_t ot al ") )
  . l i mi t ( 5)
  . expl ai n( )

# i n Pyt hon f l i ght
Dat a2015\
  . groupBy( "DEST_COUNTRY_NAME") \
  . sum( "count ") \
  . wi t hCol umnRenamed( "sum( count ) ", "dest i nat i on_t ot al ") \
  . sort ( desc( "dest i nat i on_t ot al ") ) \
  . l i mi t ( 5) \
```

```
. expl ai n( )

== Physi cal Pl an ==
TakeOrderedAndProj ect ( l i mi t =5, orderBy=[ dest i nat i on_t ot al #16194L DESC] , out pu. . .
+- *HashAggregat e( keys=[ DEST_COUNTRY_NAME#7323] , f unct i ons=[ sum( count #7325L) ] )
  +- Exchange hashpart i t i oni ng( DEST_COUNTRY_NAME#7323, 5)
    +- *HashAggregat e( keys=[ DEST_COUNTRY_NAME#7323] , f unct i ons=[ part i al _sum. . .        +- I
nMemoryTabl eScan [ DEST_COUNTRY_NAME#7323, count #7325L]
          +- I nMemoryRel at i on [ DEST_COUNTRY_NAME#7323, ORI GI N_COUNTRY_NA. . .
             +- *Scan csv [ DEST_COUNTRY_NAME#7578, ORI GI N_COUNTRY_NAME. . .
```

Although this explain plan doesn't match our exact "conceptual plan," all of the pieces are there. You can see the l i mi t statement as well as the orderBy (in the first line). You can also see how our aggregation happens in two phases, in the part i al _sum calls. This is because summing a list of numbers is commutative, and Spark can perform the sum, partition by partition. Of course we can see how we read in the DataFrame, as well.

Naturally, we don't always need to collect the data. We can also write it out to any data source that Spark supports. For instance, suppose we want to store the information in a database like PostgreSQL or write them out to another file.

Conclusion

This chapter introduced the basics of Apache Spark. We talked about transformations and actions, and how Spark lazily executes a DAG of transformations in order to optimize the execution plan on DataFrames. We also discussed how data is organized into partitions and set the stage for working with more complex transformations. In Chapter 3 we take you on a tour of the vast Spark ecosystem and look at some more advanced concepts and tools that are available in Spark, from streaming to machine learning.

Chapter 3. A Tour of Spark's Toolset

In Chapter 2, we introduced Spark's core concepts, like transformations and actions, in the context of Spark's Structured APIs. These simple conceptual building blocks are the foundation of Apache Spark's vast ecosystem of tools and libraries (Figure 3-1). Spark is composed of these primitives—the lower-level APIs and the Structured APIs—and then a series of standard libraries for additional functionality.

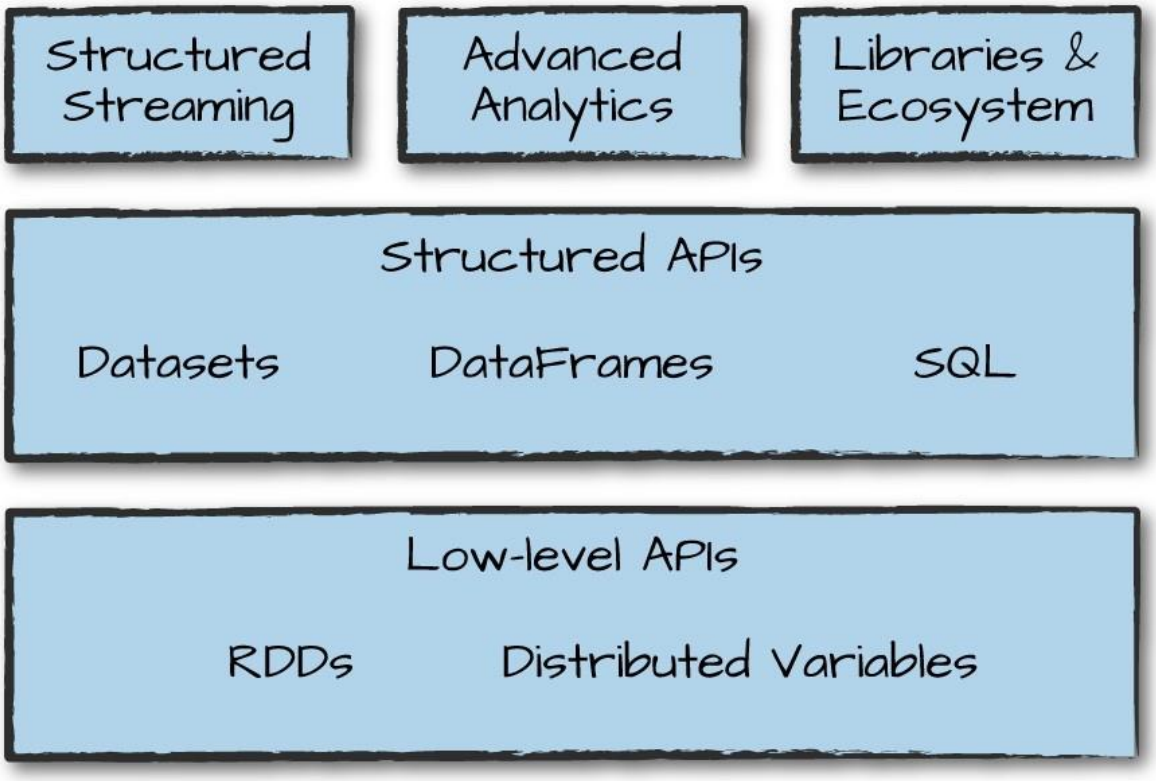


Figure 3-1. Spark's toolset

Spark's libraries support a variety of different tasks, from graph analysis and machine learning to streaming and integrations with a host of computing and storage systems. This chapter presents a whirlwind tour of much of what Spark has to offer, including some of the APIs we have not yet covered and a few of the main libraries. For each section, you will find more detailed information in other parts of this book; our purpose here is provide you with an overview of what's possible.

This chapter covers the following:

- Running production applications with spark- submi t
- Datasets: type-safe APIs for structured data
- Structured Streaming
- Machine learning and advanced analytics
- Resilient Distributed Datasets (RDD): Spark's low level APIs
- SparkR
- The third-party package ecosystem

After you've taken the tour, you'll be able to jump to the corresponding parts of the book to find answers to your questions about particular topics.

Running Production Applications

Spark makes it easy to develop and create big data programs. Spark also makes it easy to turn your interactive exploration into production applications with spark- submi t , a built-in command-line tool. spark- submi t does one thing: it lets you send your application code to a cluster and launch it to execute there. Upon submission, the application will run until it exits (completes the task) or encounters an error. You can do this with all of Spark's support cluster managers including Standalone, Mesos, and YARN.

spark- submi t offers several controls with which you can specify the resources your application needs as well as how it should be run and its command-line arguments.

You can write applications in any of Spark's supported languages and then submit them for execution. The simplest example is running an application on your local machine. We'll show this by running a sample Scala application that comes with Spark, using the following command in the directory where you downloaded Spark:

```
. /bi n/spark- submi t \
- - cl ass org. apache. spark. exampl es. SparkPi \
- - mast er l ocal \
  . /exampl es/j ars/spark- exampl es_2. 11- 2. 2. 0. j ar 10
```

This sample application calculates the digits of pi to a certain level of estimation. Here, we've told spark- submi t that we want to run on our local machine, which class and which JAR we would like to run, and some command-line arguments for that class.

We can also run a Python version of the application using the following command:

```
. /bi n/spark- submi t \
- - mast er l ocal \
  . /exampl es/src/mai n/pyt hon/pi . py 10
```

By changing the mast er argument of spark- submi t , we can also submit the same application to a cluster running Spark's standalone cluster manager, Mesos or YARN.

spark- submi t will come in handy to run many of the examples we've packaged with this book. In the rest of this chapter, we'll go through examples of some APIs that we haven't yet seen in our introduction to Spark.

Datasets: Type-Safe Structured APIs

The first API we'll describe is a type-safe version of Spark's structured API called *Datasets*, for writing statically typed code in Java and Scala. The Dataset API is not available in Python and R, because those languages are dynamically typed.

Recall that DataFrames, which we saw in the previous chapter, are a distributed collection of objects of type Row that can hold various types of tabular data. The Dataset API gives users the ability to assign a Java/Scala class to the records within a DataFrame and manipulate it as a collection of typed objects, similar to a Java ArrayLi st or Scala Seq. The APIs available on Datasets are *type-safe*, meaning that you cannot accidentally view the objects in a Dataset as being of another class than the class you put in initially. This makes Datasets especially attractive for writing large applications, with which multiple software engineers must interact through well-defined interfaces.

The Dat aset class is parameterized with the type of object contained inside: Dat aset <T> in Java and Dat aset [T] in Scala. For example, a Dat aset [Person] will be guaranteed to contain objects of class Person. As of Spark 2.0, the supported types are classes following the JavaBean pattern in Java and case classes in Scala. These types are restricted because Spark needs to be able to automatically analyze the type T and create an appropriate schema for the tabular data within your Dataset.

One great thing about Datasets is that you can use them only when you need or want to. For instance, in the following example, we'll define our own data type and manipulate it via arbitrary map and filter functions. After we've performed our manipulations, Spark can automatically turn it back into a DataFrame, and we can manipulate it further by using the hundreds of functions that Spark includes. This makes it easy to drop down to lower level, perform type-safe coding when necessary, and move higher up to SQL for more rapid analysis. Here is a small example showing how you can use both type-safe functions and DataFrame-like SQL expressions to quickly write business logic:

```
// i n Scal a
case  cl  ass  Fl  i  ght  (  DEST_COUNTRY_NAME:  St  ri  ng,
ORI GI N_COUNTRY_NAME: St ri ng,                count : Bi
gI nt ) val f l i ght sDF = spark. read
  . parquet ( "/dat a/f l i ght - dat a/parquet /2010- summary. parquet /") val f l i
ght s = f l i ght sDF. as[ Fl i ght ]
```

One final advantage is that when you call col l ect or t ake on a Dataset, it will collect objects of the proper type in your Dataset, not DataFrame Rows. This makes it easy to get type safety and securely perform manipulation in a distributed and a local manner without code changes:

```
// i n Scal a f l i
ght s
  . f i l t er( f l i ght _row => f l i ght _row. ORI GI N_COUNTRY_NAME ! = "Canada")
  . map( f l i ght _row => f l i ght _row)
  . t ake( 5)

f l i ght s
  . t ake( 5)
  . f i l t er( f l i ght _row => f l i ght _row. ORI GI N_COUNTRY_NAME ! = "Canada")
  . map( f r => Fl i ght ( f r. DEST_COUNTRY_NAME, f r. ORI GI N_COUNTRY_NAME, f r. count + 5) ) We
```

cover Datasets in depth in Chapter 11.

Structured Streaming

Structured Streaming is a high-level API for stream processing that became production-ready in Spark 2.2. With Structured Streaming, you can take the same operations that you perform in batch mode using Spark's structured APIs and run them in a streaming fashion. This can reduce latency and allow for incremental processing. The best thing about Structured Streaming is that it allows you to rapidly and quickly extract value out of streaming systems with virtually no code changes. It also makes it easy to conceptualize because you can write your batch job as a way to prototype it and then you can convert it to a streaming job. The way all of this works is by incrementally processing that data.

Let's walk through a simple example of how easy it is to get started with Structured Streaming. For this, we will use a retail dataset, one that has specific dates and times for us to be able to use. We will use the "by-day" set of files, in which one file represents one day of data.

We put it in this format to simulate data being produced in a consistent and regular manner by a different process. This is retail data so imagine that these are being produced by retail stores and sent to a location where they will be read by our Structured Streaming job.

It's also worth sharing a sample of the data so you can reference what the data looks like:

```
I nvoi ceNo, St ockCode, Descri pt i on, Quant i t y, I nvoi ceDat e, Uni t Pri ce, Cust omerI D, Count ry
536365, 85123A, WHI TE HANGI NG HEART T- LI GHT HOLDER, 6, 2010- 12- 01 08: 26: 00, 2. 55, 17. . .
536365, 71053, WHI TE METAL LANTERN, 6, 2010- 12- 01 08: 26: 00, 3. 39, 17850. 0, Uni t ed Ki n. . .
536365, 84406B, CREAM CUPI D HEARTS COAT HANGER, 8, 2010- 12- 01 08: 26: 00, 2. 75, 17850. . .
```

To ground this, let's first analyze the data as a static dataset and create a DataFrame to do so.

We'll also create a schema from this static dataset (there are ways of using schema inference with streaming that we will touch on in Part V):

```
// in Scala val staticDataFrame = spark.read.format(
"csv")
  .option("header", "true")
  .option("inferSchema", "true")   .load("/data
/retail-data/by-day/*.csv")

staticDataFrame.createOrReplaceTempView("retail_data") val st
aticSchema = staticDataFrame.schema

# in Python staticDataFrame = spark.read.format
("csv") \
  .option("header", "true") \
  .option("inferSchema", "true") \   .load("/data
/retail-data/by-day/*.csv")

staticDataFrame.createOrReplaceTempView("retail_data") stat
icSchema = staticDataFrame.schema
```

Because we're working with time–series data, it's worth mentioning how we might go along grouping and aggregating our data. In this example we'll take a look at the sale hours during which a given customer (identified by CustomerId) makes a large purchase. For example, let's add a total cost column and see on what days a customer spent the most.

The window function will include all data from each day in the aggregation. It's simply a window over the time–series column in our data. This is a helpful tool for manipulating date and timestamps because we can specify our requirements in a more human form (via intervals), and Spark will group all of them together for us:

```
// in Scala import org.apache.spark.sql.functions.{ window, column, desc,
col } staticDataFrame   .selectExpr(
   "CustomerId",
   "(UnitPrice * Quantity) as total_cost ",
   "InvoiceDate")   .groupBy(    col("CustomerId"), window( col("I
nvoiceDate"), "1 day") )
  .sum("total_cost ")
  .show(5)

# in Python from pyspark.sql.functions import window, column,
desc, col staticDataFrame\   .selectExpr(
   "CustomerId",
   "(UnitPrice * Quantity) as total_cost ",
   "InvoiceDate") \
  .groupBy(    col("CustomerId"), window( col("InvoiceDate"), "1 day")
) \
```

```
  . sum( "t ot al _cost ") \
  . show( 5)
```

It's worth mentioning that you can also run this as SQL code, just as we saw in the previous chapter.

Here's a sample of the output that you'll see:

```
+- - - - - - - - - - +- - - - - - - - - - - - - - - - - - - +- - - - - - - - - - - - - - - - - +
|Cust omerI d|            wi ndow|   sum( t ot al _cost ) |
+- - - - - - - - - - +- - - - - - - - - - - - - - - - - - - +- - - - - - - - - - - - - - - - - +
|   17450. 0|[ 2011- 09- 20 00: 00. . . |         71601. 44| . . .
|     nul l |[ 2011- 12- 08 00: 00. . . |31975. 590000000007|
+- - - - - - - - - - +- - - - - - - - - - - - - - - - - - - +- - - - - - - - - - - - - - - - - +
```

The null values represent the fact that we don't have a cust omerI d for some transactions.

That's the static DataFrame version; there shouldn't be any big surprises in there if you're familiar with the syntax.

Because you're likely running this in local mode, it's a good practice to set the number of shuffle partitions to something that's going to be a better fit for local mode. This configuration specifies the number of partitions that should be created after a shuffle. By default, the value is 200, but because there aren't many executors on this machine, it's worth reducing this to 5. We did this same operation in Chapter 2, so if you don't remember why this is important, feel free to flip back to review. spark. conf . set ("spark. sql . shuf f l e. part i t i ons", "5")

Now that we've seen how that works, let's take a look at the streaming code! You'll notice that very little actually changes about the code. The biggest change is that we used readStream instead of read, additionally you'll notice the maxFilesPerTrigger option, which simply specifies the number of files we should read in at once. This is to make our demonstration more "streaming," and in a production scenario this would probably be omitted.

```
val st reami ngDat aFrame = spark. readSt ream
  . schema( st at i cSchema)
  . opt i on( "maxFi l esPerTri gger", 1)
  . f ormat ( "csv")
  . opt i on( "header", "t rue")
  . l oad( "/dat a/ret ai l - dat a/by- day/*. csv")

# i n Pyt hon st reami ngDat aFrame = spark.
readSt ream\    . schema( st at i cSchema) \
  . opt i on( "maxFi l esPerTri gger", 1) \
  . f ormat ( "csv") \
  . opt i on( "header", "t rue") \
```

```
  .load("/data/retail-data/by-day/*.csv")
```

Now we can see whether our DataFrame is streaming:

```
streamingDataFrame.isStreaming // returns true
```

Let's set up the same business logic as the previous DataFrame manipulation. We'll perform a summation in the process:

```
// in Scala
val purchaseByCustomerPerHour = streamingDataFrame
  .selectExpr(
    "CustomerId",
    "(UnitPrice * Quantity) as total_cost",
    "InvoiceDate")
  .groupBy(
    $"CustomerId", window( $"InvoiceDate", "1 day") )
  .sum( "total_cost")

# in Python purchaseByCustomerPerHour = streamingDat
aFrame\
  .selectExpr(
    "CustomerId",
    "(UnitPrice * Quantity) as total_cost",
    "InvoiceDate") \   .groupBy(     col ( "CustomerId") , window( col ( "I
nvoiceDate") , "1 day") ) \   .sum( "total_cost")
```

This is still a lazy operation, so we will need to call a streaming action to start the execution of this data flow.

Streaming actions are a bit different from our conventional static action because we're going to be populating data somewhere instead of just calling something like count (which doesn't make any sense on a stream anyways). The action we will use will output to an in-memory table that we will update after each *trigger*. In this case, each trigger is based on an individual file (the read option that we set). Spark will mutate the data in the in-memory table such that we will always have the highest value as specified in our previous aggregation:

```
// in Scala purchaseByCustomerPerHour.writ
eStream
  .format ( "memory") // memory = store in-memory table
  .queryName( "customer_purchases") // the name of the in-memory table
  .outputMode( "complete") // complete = all the counts should be in the table
  .start ( )

# in Python
```

```
purchaseByCust omerPerHour. wri t eSt ream\
   . f ormat ( "memory") \
   . queryName( "cust omer_purchases") \
   . out put Mode( "compl et e") \
   . st art ( )
```

When we start the stream, we can run queries against it to debug what our result will look like if we were to write this out to a production sink:

```
// i n Scal a spark. sql
( """
 SELECT *
 FROM cust omer_purchases
 ORDER BY ` sum( t ot al _cost ) ` DESC
 """)
 . show( 5)

# i n Pyt hon spark.
sql ( """
 SELECT *
 FROM cust omer_purchases
 ORDER BY ` sum( t ot al _cost ) ` DESC
 """) \
 . show( 5)
```

You'll notice that the composition of our table changes as we read in more data! With each file, the results might or might not be changing based on the data. Naturally, because we're grouping customers, we hope to see an increase in the top customer purchase amounts over time (and do for a period of time!). Another option you can use is to write the results out to the console:

```
purchaseByCust omerPerHour. wri t eSt ream    .
f ormat ( "consol e")
   . queryName( "cust omer_purchases_2")
   . out put Mode( "compl et e")
   . st art ( )
```

You shouldn't use either of these streaming methods in production, but they do make for convenient demonstration of Structured Streaming's power. Notice how this window is built on event time, as well, not the time at which Spark processes the data. This was one of the shortcomings of Spark Streaming that Structured Streaming has resolved. We cover Structured Streaming in depth in Part V.

Machine Learning and Advanced Analytics

Another popular aspect of Spark is its ability to perform large-scale machine learning with a built-in library of machine learning algorithms called MLlib. MLlib allows for preprocessing, munging, training of models, and making predictions at scale on data. You can even use models trained in MLlib to make predictions in Strucutred Streaming. Spark provides a sophisticated machine learning API for performing a variety of machine learning tasks, from classification to regression, and clustering to deep learning. To demonstrate this functionality, we will perform some basic clustering on our data using a standard algorithm called ǒ-means.

WHAT IS *K*-MEANS?

ǒ-means is a clustering algorithm in which "ǒ" centers are randomly assigned within the data. The points closest to that point are then "assigned" to a class and the center of the assigned points is computed. This center point is called the *centroid*. We then label the points closest to that centroid, to the centroid's class, and shift the centroid to the new center of that cluster of points. We repeat this process for a finite set of iterations or until convergence (our center points stop changing).

Spark includes a number of preprocessing methods out of the box. To demonstrate these methods, we will begin with some raw data, build up transformations before getting the data into the right format, at which point we can actually train our model and then serve predictions:

```
st at i cDat aFrame. pri nt Schema( )
```

```
root
 |- - I nvoi ceNo: st ri ng ( nul l abl e = t rue)
 |- - St ockCode: st ri ng ( nul l abl e = t rue)
 |- - Descri pt i on: st ri ng ( nul l abl e = t rue)
 |- - Quant i t y: i nt eger ( nul l abl e = t rue)
 |- - I nvoi ceDat e: t i mest amp ( nul l abl e = t rue)
 |- - Uni t Pri ce: doubl e ( nul l abl e = t rue)
 |- - Cust omerI D: doubl e ( nul l abl e = t rue)
 |- - Count ry: st ri ng ( nul l abl e = t rue)
```

Machine learning algorithms in MLlib require that data is represented as numerical values. Our current data is represented by a variety of different types, including timestamps, integers, and strings. Therefore we need to transform this data into some numerical representation. In this instance, we'll use several DataFrame transformations to manipulate our date data:

```
// i n Scal a i mport org. apache. spark. sql . f unct i ons. dat
e_f ormat val preppedDat aFrame = st at i cDat aFrame
 . na. f i l l ( 0)
 . wi t hCol umn( "day_of _week", dat e_f ormat ( $"I nvoi ceDat e", "EEEE") )
 . coal esce( 5)
```

```
# in Python from pyspark.sql.functions import date_format, col
preppedDataFrame = staticDataFrame\
  .na.fill(0)\
  .withColumn("day_of_week", date_format(col("InvoiceDate"), "EEEE"))\
  .coalesce(5)
```

We are also going to need to split the data into training and test sets. In this instance, we are going to do this manually by the date on which a certain purchase occurred; however, we could also use MLlib's transformation APIs to create a training and test set via train validation splits or cross validation (these topics are covered at length in Part VI):

```
// in Scala
val trainDataFrame = preppedDataFrame
  .where("InvoiceDate < '2011-07-01'")
val testDataFrame = preppedDataFrame
  .where("InvoiceDate >= '2011-07-01'")

# in Python trainDataFrame = preppedDataFrame\
  .where("InvoiceDate < '2011-07-01'") testDataFrame = preppedDataFrame\
  .where("InvoiceDate >= '2011-07-01'")
```

Now that we've prepared the data, let's split it into a training and test set. Because this is a time– series set of data, we will split by an arbitrary date in the dataset. Although this might not be the optimal split for our training and test, for the intents and purposes of this example it will work just fine. We'll see that this splits our dataset roughly in half:

```
trainDataFrame.count()
testDataFrame.count()
```

Note that these transformations are DataFrame transformations, which we cover extensively in Part II. Spark's MLlib also provides a number of transformations with which we can automate some of our general transformations. One such transformer is a StringIndexer:

```
// in Scala import org.apache.spark.ml.feature.StringIndexer
indexer = new StringIndexer()  .setInputCol("day_of_week")
  .setOutputCol("day_of_week_index")

# in Python from pyspark.ml.feature import StringIndexer
indexer = StringIndexer()\
  .setInputCol("day_of_week")\
  .setOutputCol("day_of_week_index")
```

This will turn our days of weeks into corresponding numerical values. For example, Spark might represent Saturday as 6, and Monday as 1. However, with this numbering scheme, we are

implicitly stating that Saturday is greater than Monday (by pure numerical values). This is obviously incorrect. To fix this, we therefore need to use a OneHot Encoder to encode each of these values as their own column. These Boolean flags state whether that day of week is the relevant day of the week:

```
// i n Scal a i mport org. apache. spark. ml . f eat ure. OneHot
Encoder val encoder = new OneHot Encoder( )   . set I nput
Col ( "day_of _week_i ndex")
 . set Out put Col ( "day_of _week_encoded")

# i n Pyt hon f rom pyspark. ml . f eat ure i mport OneHot
Encoder encoder = OneHot Encoder( ) \
 . set I nput Col ( "day_of _week_i ndex") \
 . set Out put Col ( "day_of _week_encoded")
```

Each of these will result in a set of columns that we will "assemble" into a vector. All machine learning algorithms in Spark take as input a Vect or type, which must be a set of numerical values:

```
// i n Scal a i mport org. apache. spark. ml . f eat ure. Vect
orAssembl er

val vect orAssembl er = new Vect orAssembl er( )
 . set I nput Col s( Array( "Uni t Pri ce", "Quant i t y", "day_of _week_encoded") )
 . set Out put Col ( "f eat ures")

# i n Pyt hon f rom pyspark. ml . f eat ure i mport Vect
orAssembl er

vect orAssembl er = Vect orAssembl er( ) \
 . set I nput Col s( [ "Uni t Pri ce", "Quant i t y", "day_of _week_encoded"] ) \   . set
Out put Col ( "f eat ures")
```

Here, we have three key features: the price, the quantity, and the day of week. Next, we'll set this up into a pipeline so that any future data we need to transform can go through the exact same process:

```
// i n Scal a i mport org. apache. spark. ml .
Pi pel i ne

val t ransf ormat i onPi pel i ne = new Pi pel i ne( )
 . set St ages( Array( i ndexer, encoder, vect orAssembl er) )

# i n Pyt hon
f rom pyspark. ml i mport Pi pel i ne

t ransf ormat i onPi pel i ne = Pi pel i ne( ) \
```

```
  .setStages([indexer, encoder, vectorAssembler])
```

Preparing for training is a two-step process. We first need to fit our transformers to this dataset.
We cover this in depth in Part VI, but basically our StringIndexer needs to know how many unique values there are to be indexed. After those exist, encoding is easy but Spark must look at all the distinct values in the column to be indexed in order to store those values later on:

```
// in Scala
val fittedPipeline = transformationPipeline.fit(trainDataFrame)

# in Python
fittedPipeline = transformationPipeline.fit(trainDataFrame)
```

After we fit the training data, we are ready to take that fitted pipeline and use it to transform all of our data in a consistent and repeatable way:

```
// in Scala
val transformedTraining = fittedPipeline.transform(trainDataFrame)

# in Python
transformedTraining = fittedPipeline.transform(trainDataFrame)
```

At this point, it's worth mentioning that we could have included our model training in our pipeline. We chose not to in order to demonstrate a use case for caching the data. Instead, we're going to perform some hyperparameter tuning on the model because we do not want to repeat the exact same transformations over and over again; specifically, we'll use caching, an optimization that we discuss in more detail in Part IV. This will put a copy of the intermediately transformed dataset into memory, allowing us to repeatedly access it at much lower cost than running the entire pipeline again. If you're curious to see how much of a difference this makes, skip this line and run the training without caching the data. Then try it after caching; you'll see the results are significant: `transformedTraining.cache()`

We now have a training set; it's time to train the model. First we'll import the relevant model that we'd like to use and instantiate it:

```
// in Scala
import org.apache.spark.ml.clustering.KMeans
val kmeans = new KMeans()
  .setK(20)
  .setSeed(1L)

# in Python
from pyspark.ml.clustering import KMeans
kmeans = KMeans().\
  setK(20).\
```

```
  .setSeed(1L)
```

In Spark, training machine learning models is a two-phase process. First, we initialize an untrained model, and then we train it. There are always two types for every algorithm in MLlib's DataFrame API. They follow the naming pattern of Algorithm, for the untrained version, and AlgorithmModel for the trained version. In our example, this is KMeans and then KMeansModel .

Estimators in MLlib's DataFrame API share roughly the same interface that we saw earlier with our preprocessing transformers like the StringIndexer. This should come as no surprise because it makes training an entire pipeline (which includes the model) simple. For our purposes here, we want to do things a bit more step by step, so we chose to not do this in this example:

```
// in Scala
val kmModel = kmeans.fit(transformedTraining)

# in Python
kmModel = kmeans.fit(transformedTraining)
```

After we train this model, we can compute the cost according to some success merits on our training set. The resulting cost on this dataset is actually quite high, which is likely due to the fact that we did not properly preprocess and scale our input data, which we cover in depth in Chapter 25: `kmModel.computeCost(transformedTraining)`

```
// in Scala
val transformedTest = fittedPipeline.transform(testDataFrame)

# in Python
transformedTest = fittedPipeline.transform(testDataFrame)

kmModel.computeCost(transformedTest)
```

Naturally, we could continue to improve this model, layering more preprocessing as well as performing hyperparameter tuning to ensure that we're getting a good model. We leave that discussion for Part VI.

Lower-Level APIs

Spark includes a number of lower-level primitives to allow for arbitrary Java and Python object manipulation via Resilient Distributed Datasets (RDDs). Virtually everything in Spark is built on top of RDDs. As we will discuss in Chapter 4, DataFrame operations are built on top of RDDs and

compile down to these lower-level tools for convenient and extremely efficient distributed execution. There are some things that you might use RDDs for, especially when you're reading or manipulating raw data, but for the most part you should stick to the Structured APIs. RDDs are lower level than DataFrames because they reveal physical execution characteristics (like partitions) to end users.

One thing that you might use RDDs for is to parallelize raw data that you have stored in memory on the driver machine. For instance, let's parallelize some simple numbers and create a DataFrame after we do so. We then can convert that to a DataFrame to use it with other DataFrames:

```
// i n Scal a
spark. sparkCont ext . paral l el i ze( Seq( 1, 2, 3) ) . t oDF( )

# i n Pyt hon f rom pyspark. sql i mport Row spark. sparkCont ext . paral l el i ze(
[ Row( 1) , Row( 2) , Row( 3) ] ) . t oDF( )
```

RDDs are available in Scala as well as Python. However, they're not equivalent. This differs from the DataFrame API (where the execution characteristics are the same) due to some underlying implementation details. We cover lower-level APIs, including RDDs in Part IV. As end users, you shouldn't need to use RDDs much in order to perform many tasks unless you're maintaining older Spark code. There are basically no instances in modern Spark, for which you should be using RDDs instead of the structured APIs beyond manipulating some very raw unprocessed and unstructured data.

SparkR

SparkR is a tool for running R on Spark. It follows the same principles as all of Spark's other language bindings. To use SparkR, you simply import it into your environment and run your code. It's all very similar to the Python API except that it follows R's syntax instead of Python. For the most part, almost everything available in Python is available in SparkR:

```
# i n R l i brary( SparkR) sparkDF <- read. df ( "/dat a/f l i ght - dat a/csv/2015-
summary. csv",         source = "csv", header="t rue", i nf erSchema = "t rue")
t ake( sparkDF, 5)

# i n R
col l ect ( orderBy( sparkDF, "count ") , 20)
```

R users can also use other R libraries like the pipe operator in magrittr to make Spark transformations a bit more R-like. This can make it easy to use with other libraries like ggplot for more sophisticated plotting:

```
# in R library( magrittr) sparkDF %>%
orderBy( desc( sparkDF$count ) ) %>%
groupBy( "ORIGIN_COUNTRY_NAME") %>%
count ( ) %>%
 limit ( 10) %>% collect ( )
```

We will not include R code samples as we do in Python, because almost every concept throughout this book that applies to Python also applies to SparkR. The only difference will by syntax. We cover SparkR and sparklyr in Part VII.

Spark's Ecosystem and Packages

One of the best parts about Spark is the ecosystem of packages and tools that the community has created. Some of these tools even move into the core Spark project as they mature and become widely used. As of this writing, the list of packages is rather long, numbering over 300—and more are added frequently. You can find the largest index of Spark Packages at sparkpackages.org, where any user can publish to this package repository. There are also various other projects and packages that you can find on the web; for example, on GitHub.

Conclusion

We hope this chapter showed you the sheer variety of ways in which you can apply Spark to your own business and technical challenges. Spark's simple, robust programming model makes it easy to apply to a large number of problems, and the vast array of packages that have crept up around it, created by hundreds of different people, are a true testament to Spark's ability to robustly tackle a number of business problems and challenges. As the ecosystem and community grows, it's likely that more and more packages will continue to crop up. We look forward to seeing what the community has in store!

The rest of this book will provide deeper dives into the product areas in Figure 3-1.

You may read the rest of the book any way that you prefer, we find that most people hop from area to area as they hear terminology or want to apply Spark to certain problems they're facing.

Part II. Structured APIs—DataFrames, SQL, and Datasets

Chapter 4. Structured API Overview

This part of the book will be a deep dive into Spark's Structured APIs. The Structured APIs are a tool for manipulating all sorts of data, from unstructured log files to semi-structured CSV files and highly structured Parquet files. These APIs refer to three core types of distributed collection APIs:

- Datasets
- DataFrames
- SQL tables and views

Although they are distinct parts of the book, the majority of the Structured APIs apply to both *batch* and *streaming* computation. This means that when you work with the Structured APIs, it should be simple to migrate from batch to streaming (or vice versa) with little to no effort. We'll cover streaming in detail in Part V.

The Structured APIs are the fundamental abstraction that you will use to write the majority of your data flows. Thus far in this book, we have taken a tutorial-based approach, meandering our way through much of what Spark has to offer. This part offers a more in-depth exploration. In this chapter, we'll introduce the fundamental concepts that you should understand: the typed and untyped APIs (and their differences); what the core terminology is; and, finally, how Spark actually takes your Structured API data flows and executes it on the cluster. We will then provide more specific task-based information for working with certain types of data or data sources.

NOTE

Before proceeding, let's review the fundamental concepts and definitions that we covered in Part I. Spark is a distributed programming model in which the user specifies *transformations*. Multiple transformations build up a directed acyclic graph of instructions. An action begins the process of executing that graph of instructions, as a single job, by breaking it down into stages and tasks to execute across the cluster. The logical structures that we manipulate with transformations and actions are DataFrames and Datasets. To create a new DataFrame or Dataset, you call a transformation. To start computation or convert to native language types, you call an action.

DataFrames and Datasets

Part I discussed DataFrames. Spark has two notions of structured collections: DataFrames and Datasets. We will touch on the (nuanced) differences shortly, but let's define what they both represent first.

DataFrames and Datasets are (distributed) table-like collections with well-defined rows and columns. Each column must have the same number of rows as all the other columns (although you can use nul l to specify the absence of a value) and each column has type information that must be consistent for every row in the collection. To Spark, DataFrames and Datasets represent immutable, lazily evaluated plans that specify what operations to apply to data residing at a location to generate some output. When we perform an action on a DataFrame, we instruct Spark to perform the actual transformations and return the result. These represent plans of how to manipulate rows and columns to compute the user's desired result.

> **NOTE**
>
> Tables and views are basically the same thing as DataFrames. We just execute SQL against them instead of DataFrame code. We cover all of this in Chapter 10, which focuses specifically on Spark SQL.

To add a bit more specificity to these definitions, we need to talk about schemas, which are the way you define the types of data you're storing in this distributed collection.

Schemas

A schema defines the column names and types of a DataFrame. You can define schemas manually or read a schema from a data source (often called *schema on read*). Schemas consist of types, meaning that you need a way of specifying what lies where.

Overview of Structured Spark Types

Spark is effectively a programming language of its own. Internally, Spark uses an engine called *Catalyst* that maintains its own type information through the planning and processing of work. In doing so, this opens up a wide variety of execution optimizations that make significant differences. Spark types map directly to the different language APIs that Spark maintains and there exists a lookup table for each of these in Scala, Java, Python, SQL, and R. Even if we use Spark's Structured APIs from Python or R, the majority of our manipulations will operate strictly on *Spark types*, not Python types. For example, the following code does not perform addition in Scala or Python; it actually performs addition *purely in Spark*:

```
// in Scala val df = spark.range(500).toDF(
"number").select(df.col("number") + 10)

# in Python df = spark.range(500).toDF(
"number").select(df["number"] + 10)
```

This addition operation happens because Spark will convert an expression written in an input language to Spark's internal Catalyst representation of that same type information. It then will operate on that internal representation. We touch on why this is the case momentarily, but before we can, we need to discuss Datasets.

DataFrames Versus Datasets

In essence, within the Structured APIs, there are two more APIs, the "untyped" DataFrames and the "typed" Datasets. To say that DataFrames are untyped is aslightly inaccurate; they have types, but Spark maintains them completely and only checks whether those types line up to those specified in the schema at *runtime*. Datasets, on the other hand, check whether types conform to the specification at *compile time*. Datasets are only available to Java Virtual Machine (JVM)– based languages (Scala and Java) and we specify types with case classes or Java beans.

For the most part, you're likely to work with DataFrames. To Spark (in Scala), DataFrames are simply Datasets of Type Row. The "Row" type is Spark's internal representation of its optimized in-memory format for computation. This format makes for highly specialized and efficient computation because rather than using JVM types, which can cause high garbage-collection and object instantiation costs, Spark can operate on its own internal format without incurring any of those costs. To Spark (in Python or R), there is no such thing as a Dataset: everything is a DataFrame and therefore we always operate on that optimized format.

> **NOTE**
>
> The internal Catalyst format is well covered in numerous Spark presentations. Given that this book is intended for a more general audience, we'll refrain from going into the implementation. If you're curious, there are some excellent talks by Josh Rosen and Herman van Hovell, both of Databricks, about their work in the development of Spark's Catalyst engine.

Understanding DataFrames, Spark Types, and Schemas takes some time to digest. What you need to know is that when you're using DataFrames, you're taking advantage of Spark's optimized internal format. This format applies the same efficiency gains to all of Spark's language APIs. If you need strict compile-time checking, read Chapter 11 to learn more about it.

Let's move onto some friendlier and more approachable concepts: columns and rows.

Columns

Columns represent a *simple type* like an integer or string, a *complex type* like an array or map, or a *null value*. Spark tracks all of this type information for you and offers a variety of ways, with which you can transform columns. Columns are discussed extensively in Chapter 5, but for the most part you can think about Spark Col umn types as columns in a table.

Rows

A row is nothing more than a record of data. Each record in a DataFrame must be of type Row, as we can see when we collect the following DataFrames. We can create these rows manually from SQL, from Resilient Distributed Datasets (RDDs), from data sources, or manually from scratch. Here, we create one by using a range:

```
// i n Scal a spark. range( 2) . t oDF( ) .
col l ect ( )

# i n Pyt hon spark. range( 2) .
col l ect ( )
```

These both result in an array of Row objects.

Spark Types

We mentioned earlier that Spark has a large number of internal type representations. We include a handy reference table on the next several pages so that you can most easily reference what type, in your specific language, lines up with the type in Spark.

Before getting to those tables, let's talk about how we instantiate, or declare, a column to be of a certain type.

To work with the correct Scala types, use the following:

```
i mport org. apache. spark. sql . t ypes. _ val b =
Byt eType
```

To work with the correct Java types, you should use the factory methods in the following package:

```
i mport org. apache. spark. sql . t ypes. Dat aTypes; Byt
eType x = Dat aTypes. Byt eType;
```

Python types at times have certain requirements, which you can see listed in Table 4-1, as do Scala and Java, which you can see listed in Tables 4-2 and 4-3, respectively. To work with the correct Python types, use the following:

```
f rom pyspark. sql . t ypes i mport * b =
Byt eType( )
```

The following tables provide the detailed type information for each of Spark's language bindings.

Table 4-1. Python type reference

Data type	Value type in Python	API to access or create a data type
ByteType	int or long. Note: Numbers will be converted to 1-byte signed integer numbers at runtime. Ensure that numbers are within the range of –128 to 127.	ByteType()
ShortType	int or long. Note: Numbers will be converted to 2-byte signed integer numbers at runtime. Ensure that numbers are within the range of –32768 to 32767.	ShortType()
IntegerType	int or long. Note: Python has a lenient definition of "integer." Numbers that are too large will be rejected by Spark SQL if you use the IntegerType(). It's best practice to use LongType.	IntegerType()
LongType	long. Note: Numbers will be converted to 8-byte signed integer numbers at runtime. Ensure that numbers are within the range of –9223372036854775808 to 9223372036854775807. Otherwise, convert data to decimal.Decimal and use DecimalType.	LongType()
FloatType	float. Note: Numbers will be converted to 4-byte singleprecision floating-point numbers at runtime.	FloatType()
DoubleType	float	DoubleType()
DecimalType	decimal.Decimal	DecimalType()
StringType	string	StringType()
BinaryType	bytearray	BinaryType()
BooleanType	bool	BooleanType()
TimestampType	datetime.datetime	TimestampType()
DateType	datetime.date	DateType()
		ArrayType(elementType, [containsNull]). Note: The

ArrayType	list, tuple, or array	default value of containsNull is True.
MapType	dict	MapType(keyType, valueType, [valueContainsNull]). Note: The default value of valueContainsNull is True.
StructType	list or tuple	StructType(fields). Note: f i el ds is a list of StructFields. Also, fields StructFields. Also, fields with the same name are not allowed.
StructField	The value type in Python of the data type of this field (for example, Int for a StructField with the data type IntegerType)	StructField(name, dataType, [nullable]) Note: The default value of nullable is True.

Table 4-2. Scala type reference

Data type	Value type in Scala	API to access or create a data type
ByteType	Byte	ByteType
ShortType	Short	ShortType
IntegerType	Int	IntegerType
LongType	Long	LongType
FloatType	Float	FloatType
DoubleType	Double	DoubleType
DecimalType	java.math.BigDecimal	DecimalType
StringType	String	StringType
BinaryType	Array[Byte]	BinaryType

BooleanType	Boolean	BooleanType
TimestampType	java.sql.Timestamp	TimestampType
DateType	java.sql.Date	DateType
ArrayType	scala.collection.Seq	ArrayType(elementType, [containsNull]). Note: The default value of containsNull is true.
MapType	scala.collection.Map	MapType(keyType, valueType, [valueContainsNull]). Note: The default value of valueContainsNull is true.
StructType	org.apache.spark.sql.Row	StructType(fields). Note: fields is an Array of StructFields. Also, fields with the same name are not allowed.
StructField	The value type in Scala of the data type of this field (for example, Int for a StructField with the data type IntegerType)	StructField(name, dataType, [nullable]). Note: The default value of nullable is true.

Table 4-3. Java type reference

Data type	Value type in Java	API to access or create a data type
ByteType	byte or Byte	DataTypes.ByteType
ShortType	short or Short	DataTypes.ShortType
IntegerType	int or Integer	DataTypes.IntegerType
LongType	long or Long	DataTypes.LongType
FloatType	float or Float	DataTypes.FloatType
DoubleType	double or Double	DataTypes.DoubleType
DecimalType	java.math.BigDecimal	DataTypes.createDecimalType() DataTypes.createDecimalType(precision, scale).
StringType	String	DataTypes.StringType
BinaryType	byte[]	DataTypes.BinaryType
BooleanType	boolean or Boolean	DataTypes.BooleanType

TimestampType	java.sql.Timestamp	DataTypes.TimestampType
DateType	java.sql.Date	DataTypes.DateType
ArrayType	java.util.List	DataTypes.createArrayType(elementType). Note: The value of containsNull will be true DataTypes.createArrayType(elementType, containsNull).
MapType	java.util.Map	DataTypes.createMapType(keyType, valueType). Note: The value of valueContainsNull will be true. DataTypes.createMapType(keyType, valueType, valueContainsNull)
StructType		DataTypes.createStructType(fields). Note: fields is a org.apache.spark.sql.Row List or an array of StructFields. Also, two fields with the same name are not allowed.
StructField	The value type in Java of the data type of this field (for example, int for a StructField with the data type IntegerType)	DataTypes.createStructField(name, dataType, nullable)

It's worth keeping in mind that the types might change over time as Spark SQL continues to grow so you may want to reference Spark's documentation for future updates. Of course, all of these types are great, but you almost never work with purely static DataFrames. You will always manipulate and transform them. Therefore it's important that we give you an overview of the execution process in the Structured APIs.

Overview of Structured API Execution

This section will demonstrate how this code is actually executed across a cluster. This will help you understand (and potentially debug) the process of writing and executing code on clusters, so let's walk through the execution of a single structured API query from user code to executed code. Here's an overview of the steps:

1. Write DataFrame/Dataset/SQL Code.
2. If valid code, Spark converts this to a *Logical Plan*.
3. Spark transforms this *Logical Plan* to a *Physical Plan*, checking for optimizations along the way.
4. Spark then executes this *Physical Plan* (RDD manipulations) on the cluster.

To execute code, we must write code. This code is then submitted to Spark either through the console or via a submitted job. This code then passes through the Catalyst Optimizer, which

decides how the code should be executed and lays out a plan for doing so before, finally, the code is run and the result is returned to the user. Figure 4-1 shows the process.

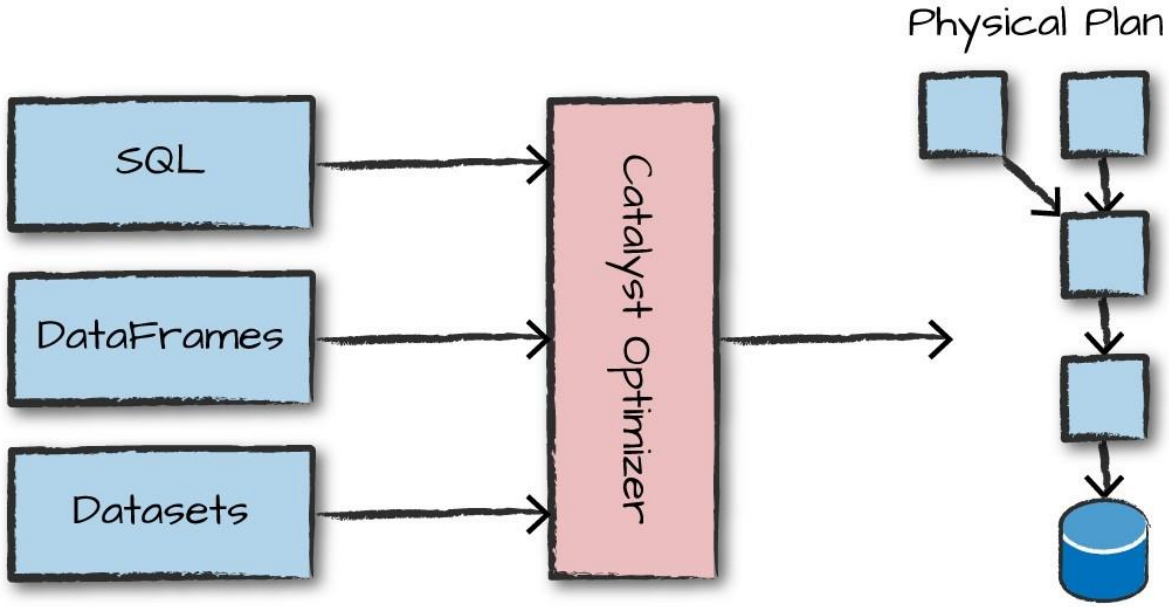


Figure 4-1. The Catalyst Optimizer

Logical Planning

The first phase of execution is meant to take user code and convert it into a logical plan. Figure 4-2 illustrates this process.

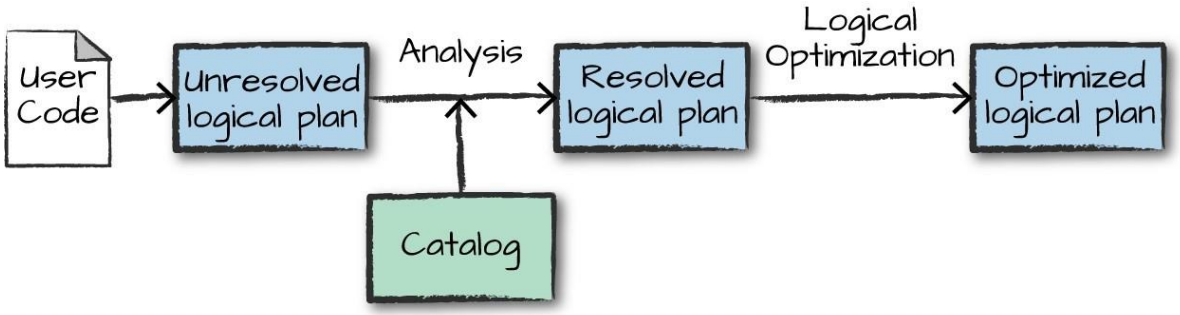


Figure 4-2. The structured API logical planning process

This logical plan only represents a set of abstract transformations that do not refer to executors or drivers, it's purely to convert the user's set of expressions into the most optimized version. It does this by converting user code into an *unresolved logical plan*. This plan is unresolved because although your code might be valid, the tables or columns that it refers to might or might not exist. Spark uses the *catalog*, a repository of all table and DataFrame information, to *resolve* columns and tables in the *analyzer*. The analyzer might reject the unresolved logical plan if the required table or column name does not exist in the catalog. If the analyzer can resolve it, the result is passed through the Catalyst Optimizer, a collection of rules that attempt

to optimize the logical plan by pushing down predicates or selections. Packages can extend the Catalyst to include their own rules for domain-specific optimizations.

Physical Planning

After successfully creating an optimized logical plan, Spark then begins the physical planning process. The *physical plan*, often called a Spark plan, specifies how the logical plan will execute on the cluster by generating different physical execution strategies and comparing them through a cost model, as depicted in Figure 4-3. An example of the cost comparison might be choosing how to perform a given join by looking at the physical attributes of a given table (how big the table is or how big its partitions are).

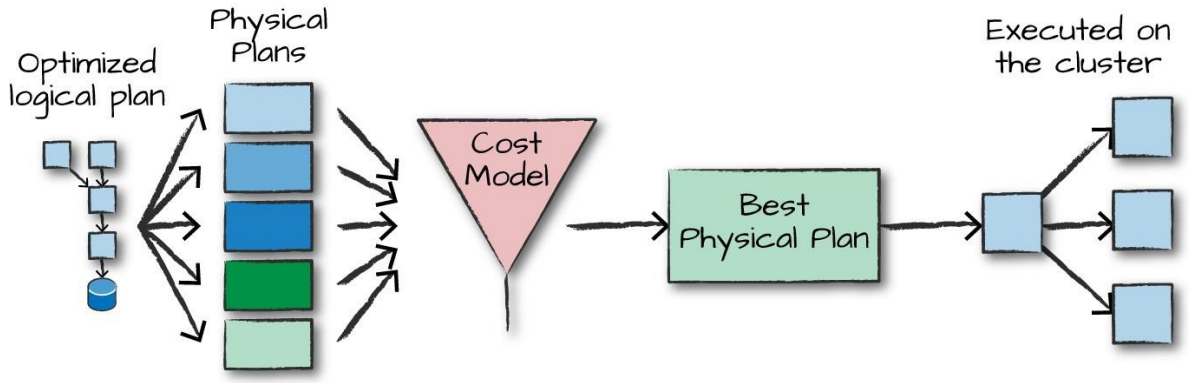


Figure 4-3. The physical planning process

Physical planning results in a series of RDDs and transformations. This result is why you might have heard Spark referred to as a compiler—it takes queries in DataFrames, Datasets, and SQL and compiles them into RDD transformations for you.

Execution

Upon selecting a physical plan, Spark runs all of this code over RDDs, the lower-level programming interface of Spark (which we cover in Part III). Spark performs further optimizations at runtime, generating native Java bytecode that can remove entire tasks or stages during execution. Finally the result is returned to the user.

Conclusion

In this chapter, we covered Spark Structured APIs and how Spark transforms your code into what will physically execute on the cluster. In the chapters that follow, we cover core concepts and how to use the key functionality of the Structured APIs.

Chapter 5. Basic Structured Operations

In Chapter 4, we introduced the core abstractions of the Structured API. This chapter moves away from the architectural concepts and toward the tactical tools you will use to manipulate DataFrames and the data within them. This chapter focuses exclusively on fundamental DataFrame operations and avoids aggregations, window functions, and joins. These are discussed in subsequent chapters.

Definitionally, a DataFrame consists of a series of *records* (like rows in a table), that are of type Row, and a number of *columns* (like columns in a spreadsheet) that represent a computation expression that can be performed on each individual record in the Dataset. *Schemas* define the name as well as the type of data in each column. *Partitioning* of the DataFrame defines the layout of the DataFrame or Dataset's physical distribution across the cluster. The *partitioning scheme* defines how that is allocated. You can set this to be based on values in a certain column or nondeterministically.

Let's create a DataFrame with which we can work:

```
// i n Scal a val df = spark. read. f ormat ( "j
son")
  . l oad( "/dat a/f l i ght - dat a/j son/2015- summary. j son")

# i n Pyt hon
df = spark. read. f ormat ( "j son") . l oad( "/dat a/f l i ght - dat a/j son/2015- summary. j son")
```

We discussed that a DataFame will have columns, and we use a schema to define them. Let's take a look at the schema on our current DataFrame: `df . pri nt Schema( )`

Schemas tie everything together, so they're worth belaboring.

Schemas

A schema defines the column names and types of a DataFrame. We can either let a data source define the schema (called *schema-on-read*) or we can define it explicitly ourselves.

> **WARNING**
>
> Deciding whether you need to define a schema prior to reading in your data depends on your use case. For ad hoc analysis, schema-on-read usually works just fine (although at times it can be a bit slow with plain-text file formats like CSV or JSON). However, this can also lead to precision issues like a l ong type incorrectly set as an integer when reading in a file. When using Spark for production Extract, Transform, and Load (ETL), it is often a

good idea to define your schemas manually, especially when working with untyped data sources like CSV and JSON because schema inference can vary depending on the type of data that you read in.

Let's begin with a simple file, which we saw in Chapter 4, and let the semi-structured nature of line-delimited JSON define the structure. This is flight data from the United States Bureau of Transportation statistics:

```
// i n Scal a
spark. read. f ormat ( "j son") . l oad( "/dat a/f l i ght - dat a/j son/2015- summary. j son") . schema
```

Scala

returns the following:

```
org. apache. spark. sql . t ypes. St ruct Type = . . .
St ruct Type( St ruct Fi el d( DEST_COUNTRY_NAME, St ri ngType, t rue) ,
St ruct Fi el d( ORI GI N_COUNTRY_NAME, St ri ngType, t rue) ,
St ruct Fi el d( count , LongType, t rue) )
```

```
# i n Pyt hon
spark. read. f ormat ( "j son") . l oad( "/dat a/f l i ght - dat a/j son/2015- summary. j son") . schema
```

Python

returns the following:

```
St ruct Type( Li st ( St ruct Fi el d( DEST_COUNTRY_NAME, St ri ngType, t rue) ,
St ruct Fi el d( ORI GI N_COUNTRY_NAME, St ri ngType, t rue) ,
St ruct Fi el d( count , LongType, t rue) ) )
```

A schema is a St ruct Type made up of a number of fields, St ruct Fi el ds, that have a name, type, a Boolean flag which specifies whether that column can contain missing or nul l values, and, finally, users can optionally specify associated metadata with that column. The metadata is a way of storing information about this column (Spark uses this in its machine learning library).

Schemas can contain other St ruct Types (Spark's complex types). We will see this in Chapter 6 when we discuss working with complex types. If the types in the data (at runtime) do not match the schema, Spark will throw an error. The example that follows shows how to create and enforce a specific schema on a DataFrame.

```
// i n Scal a i mport org. apache. spark. sql . t ypes. { St ruct Fi el d, St ruct Type, St ri ngType, LongType}
i mport org. apache. spark. sql . t ypes. Met adat a

val myManual Schema = St ruct Type( Array(
  St ruct Fi el d( "DEST_COUNTRY_NAME", St ri ngType, t rue) ,
  St ruct Fi el d( "ORI GI N_COUNTRY_NAME", St ri ngType, t rue) ,
  St ruct Fi el d( "count ", LongType, f al se,
    Met adat a. f romJ son( "{ \"hel l o\": \"worl d\"} ") )
) )
val df = spark. read. f ormat ( "j son") . schema( myManual Schema)
```

```
. l oad( "/dat a/f l i ght - dat a/j son/2015- summary. j son")
```

Here's how to do the same in Python:

```
# i n Pyt hon f rom pyspark. sql . t ypes i mport St ruct Fi el d, St ruct Type, St ri ngType, LongType

myManual Schema = St ruct Type( [
 St ruct Fi el d( "DEST_COUNTRY_NAME", St ri ngType( ) , True) ,
 St ruct Fi el d( "ORI GI N_COUNTRY_NAME", St ri ngType( ) , True) ,
 St ruct Fi el d( "count ", LongType( ) , Fal se, met adat a={ "hel l o": "worl d"} )
] ) df = spark. read. f ormat ( "j son") . schema( myManual Schema) \
 . l oad( "/dat a/f l i ght - dat a/j son/2015- summary. j son")
```

As discussed in Chapter 4, we cannot simply set types via the per-language types because Spark maintains its own type information. Let's now discuss what schemas define: columns.

Columns and Expressions

Columns in Spark are similar to columns in a spreadsheet, R dataframe, or pandas DataFrame. You can select, manipulate, and remove columns from DataFrames and these operations are represented as *expressions*.

To Spark, columns are logical constructions that simply represent a value computed on a perrecord basis by means of an expression. This means that to have a real value for a column, we need to have a row; and to have a row, we need to have a DataFrame. You cannot manipulate an individual column outside the context of a DataFrame; you must use Spark transformations within a DataFrame to modify the contents of a column.

Columns

There are a lot of different ways to construct and refer to columns but the two simplest ways are by using the col or col umn functions. To use either of these functions, you pass in a column name:

```
// i n Scal a i mport org. apache. spark. sql . f unct i ons. { col , col
umn} col ( "someCol umnName") col umn( "someCol
umnName")

# i n Pyt hon f rom pyspark. sql . f unct i ons i mport col ,
col umn col ( "someCol umnName") col umn( "someCol
umnName")
```

We will stick to using col throughout this book. As mentioned, this column might or might not exist in our DataFrames. Columns are not *resolved* until we compare the column names with those we are maintaining in the *catalog*. Column and table resolution happens in the *analyzer* phase, as discussed in Chapter 4.

NOTE

We just mentioned two different ways of referring to columns. Scala has some unique language features that allow for more shorthand ways of referring to columns. The following bits of syntactic sugar perform the exact same thing, namely creating a column, but provide no performance improvement:

```
// i n Scal a
$"myCol umn"
' myCol umn
```

The $ allows us to designate a string as a special string that should refer to an expression. The tick mark (') is a special thing called a *symbol*; this is a Scala-specific construct of referring to some identifier. They both perform the same thing and are shorthand ways of referring to columns by name.
You'll likely see all of the aforementioned references when you read different people's Spark code. We leave it to you to use whatever is most comfortable and maintainable for you and those with whom you work.

Explicit column references

If you need to refer to a specific DataFrame's column, you can use the col method on the specific DataFrame. This can be useful when you are performing a join and need to refer to a specific column in one DataFrame that might share a name with another column in the joined DataFrame. We will see this in Chapter 8. As an added benefit, Spark does not need to resolve this column itself (during the *analyzer* phase) because we did that for Spark: `df . col ( "count ")`

Expressions

We mentioned earlier that columns are expressions, but what is an expression? An *expression* is a set of transformations on one or more values in a record in a DataFrame. Think of it like a function that takes as input one or more column names, resolves them, and then potentially applies more expressions to create a single value for each record in the dataset. Importantly, this
"single value" can actually be a complex type like a Map or Array. We'll see more of the complex types in Chapter 6.

In the simplest case, an expression, created via the expr function, is just a DataFrame column reference. In the simplest case, expr("someCol ") is equivalent to col ("someCol ") .

Columns as expressions

Columns provide a subset of expression functionality. If you use col ( ) and want to perform transformations on that column, you must perform those on that column reference. When using an expression, the expr function can actually parse transformations and column references from a string and can subsequently be passed into further transformations. Let's look at some examples.

expr("someCol - 5") is the same transformation as performing col ("someCol ") - 5, or even expr("someCol ") - 5. That's because Spark compiles these to a logical tree specifying the order of operations. This might be a bit confusing at first, but remember a couple of key points:

- Columns are just expressions.
- Columns and transformations of those columns compile to the same logical plan as parsed expressions.

Let's ground this with an example:

```
( ( ( col ( "someCol ") + 5) * 200) - 6) < col ( "ot herCol ")
```

Figure 5-1 shows an overview of that logical tree.

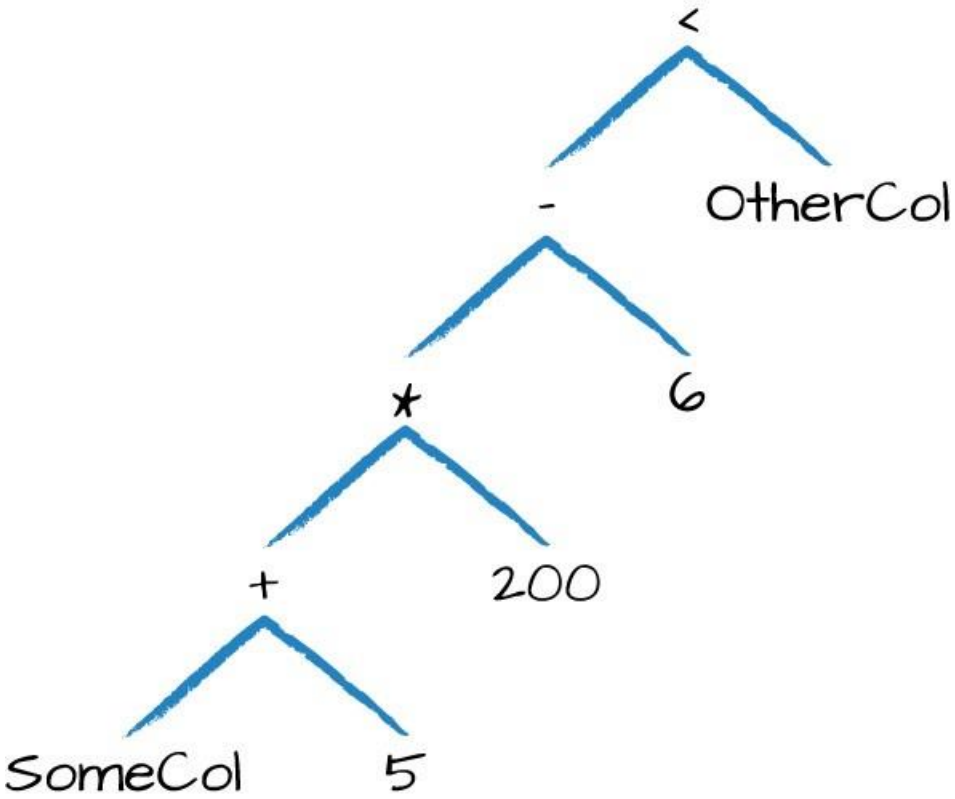


Figure 5-1. A logical tree

This might look familiar because it's a directed acyclic graph. This graph is represented equivalently by the following code:

```
// i n Scal a i mport org. apache. spark. sql . f unct i ons.
expr expr( "( ( ( someCol + 5) * 200) - 6) < ot herCol ")

# i n Pyt hon f rom pyspark. sql . f unct i ons i mport expr
expr( "( ( ( someCol + 5) * 200) - 6) < ot herCol ")
```

This is an extremely important point to reinforce. Notice how the previous expression is actually valid SQL code, as well, just like you might put in a SELECT statement? That's because this SQL expression and the previous DataFrame code compile to the same underlying logical tree prior to execution. This means that you can write your expressions as DataFrame code or as SQL expressions and get the exact same performance characteristics. This is discussed in Chapter 4.

Accessing a DataFrame's columns

Sometimes, you'll need to see a DataFrame's columns, which you can do by using something like pri nt Schema; however, if you want to programmatically access columns, you can use the col umns property to see all columns on a DataFrame:

```
spark. read. f ormat ( "j son") . l oad( "/dat a/f l i ght - dat a/j son/2015- summary. j son")   . col
umns
```

Records and Rows

In Spark, each row in a DataFrame is a single record. Spark represents this record as an object of type Row. Spark manipulates Row objects using column expressions in order to produce usable values. Row objects internally represent arrays of bytes. The byte array interface is never shown to users because we only use column expressions to manipulate them.

You'll notice commands that return individual rows to the driver will always return one or more Row types when we are working with DataFrames.

> **NOTE**
>
> We use lowercase "row" and "record" interchangeably in this chapter, with a focus on the latter. A capitalized Row refers to the Row object.

Let's see a row by calling f i rst on our DataFrame: df .

```
f i rst ( )
```

Creating Rows

You can create rows by manually instantiating a Row object with the values that belong in each column. It's important to note that only DataFrames have schemas. Rows themselves do not have schemas. This means that if you create a Row manually, you must specify the values in the same order as the schema of the DataFrame to which they might be appended (we will see this when we discuss creating DataFrames):

```
// in Scala import org.apache.spark.sql.Row
val myRow = Row("Hello", null, 1, false)

# in Python from pyspark.sql import Row
myRow = Row("Hello", None, 1, False)
```

Accessing data in rows is equally as easy: you just specify the position that you would like. In Scala or Java, you must either use the helper methods or explicitly coerce the values. However, in Python or R, the value will automatically be coerced into the correct type:

```
// in Scala myRow(0) // type Any myRow(0).
asInstanceOf[String] // String myRow.getString
ring(0) // String myRow.getInt(2) // Int

# in Python
myRow[0]
myRow[2]
```

You can also explicitly return a set of Data in the corresponding Java Virtual Machine (JVM) objects by using the Dataset APIs. This is covered in Chapter 11.

DataFrame Transformations

Now that we briefly defined the core parts of a DataFrame, we will move onto manipulating DataFrames. When working with individual DataFrames there are some fundamental objectives. These break down into several core operations, as depicted in Figure 5-2:

- We can add rows or columns
- We can remove rows or columns
- We can transform a row into a column (or vice versa)
- We can change the order of rows based on the values in columns

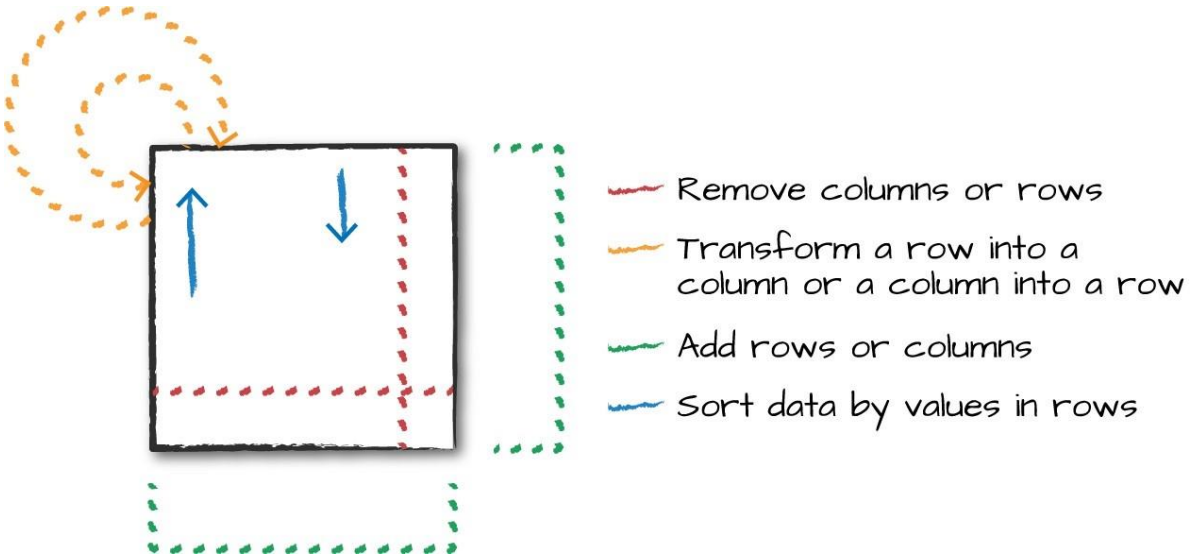


Figure 5-2. Different kinds of transformations

Luckily, we can translate all of these into simple transformations, the most common being those that take one column, change it row by row, and then return our results.

Creating DataFrames

As we saw previously, we can create DataFrames from raw data sources. This is covered extensively in Chapter 9; however, we will use them now to create an example DataFrame (for illustration purposes later in this chapter, we will also register this as a temporary view so that we can query it with SQL and show off basic transformations in SQL, as well):

```
// in Scala val df = spark.read.format("json")
  .load("/data/flight-data/json/2015-summary.json") df.
createOrReplaceTempView("dfTable")

# in Python df = spark.read.format("json").load("/data/flight-data/json/2015-summary.
json") df.createOrReplaceTempView("dfTable")
```

We can also create DataFrames on the fly by taking a set of rows and converting them to a DataFrame.

```
// in Scala import org.apache.spark.sql.Row import org.apache.spark.sql.types.{StructField,
StructType, StringType, LongType}

val myManualSchema = new StructType(Array(   new
StructField("some", StringType, true),   new StructField(
"col", StringType, true),   new StructField("names", LongType, false) )) val myRows = Seq(Row(
"Hello", null, 1L) )
```

```
val myRDD = spark. sparkCont ext . paral l el i ze( myRows) val myDf =
spark. creat eDat aFrame( myRDD, myManual Schema) myDf . show(
)
```

NOTE

In Scala, we can also take advantage of Spark's implicits in the console (and if you import them in your JAR code) by running t oDF on a Seq type. This does not play well with nul l types, so it's not necessarily recommended for production use cases.

```
// i n Scal a
val myDF = Seq( ( "Hel l o", 2, 1L) ) . t oDF( "col 1", "col 2", "col 3")

# i n Pyt hon f rom pyspark. sql i mport Row f rom pyspark. sql . t ypes i mport St ruct Fi el d, St
ruct Type, St ri ngType, LongType myManual Schema = St ruct Type( [
 St ruct Fi el d( "some", St ri ngType( ) , True) ,
 St ruct Fi el d( "col ", St ri ngType( ) , True) ,
 St ruct Fi el d( "names", LongType( ) , Fal se)
] ) myRow = Row( "Hel l o", None, 1) myDf = spark. creat eDat
aFrame( [ myRow] , myManual Schema) myDf . show( )
```

Giving an output of:

```
+- - - - -+- - - -+- - - - -+
| some| col |names|
+- - - - -+- - - -+- - - - -+ |Hel
l o|nul l |    1|
+- - - - -+- - - -+- - - - -+
```

Now that you know how to create DataFrames, let's take a look at their most useful methods that you're going to be using: the sel ect method when you're working with columns or expressions, and the sel ect Expr method when you're working with expressions in strings. Naturally some transformations are not specified as methods on columns; therefore, there exists a group of functions found in the org. apache. spark. sql . f unct i ons package.

With these three tools, you should be able to solve the vast majority of transformation challenges that you might encounter in DataFrames.

select and selectExpr

sel ect and sel ect Expr allow you to do the DataFrame equivalent of SQL queries on a table of data:

```
- - i n SQL
SELECT * FROM dat aFrameTabl e
SELECT col umnName FROM dat aFrameTabl e
```

```
SELECT col umnName * 10, ot herCol umn, someOt herCol as c FROM dat aFrameTabl e
```

In the simplest possible terms, you can use them to manipulate columns in your DataFrames. Let's walk through some examples on DataFrames to talk about some of the different ways of approaching this problem. The easiest way is just to use the sel ect method and pass in the column names as strings with which you would like to work:

```
// i n Scal a df . sel ect (
"DEST_COUNTRY_NAME") . show( 2)

# i n Pyt hon df . sel ect (
"DEST_COUNTRY_NAME") . show( 2)

- - i n SQL
SELECT DEST_COUNTRY_NAME FROM df Tabl e LI MI T 2 Giving
```

an output of:

```
+- - - - - - - - - - - - - - - - - - - +
|DEST_COUNTRY_NAME|
+- - - - - - - - - - - - - - - - +
| Uni t ed St at es|
| Uni t ed St at es|
+- - - - - - - - - - - - - - - - +
```

You can select multiple columns by using the same style of query, just add more column name strings to your sel ect method call:

```
// i n Scal a df . sel ect ( "DEST_COUNTRY_NAME", "ORI GI
N_COUNTRY_NAME") . show( 2)

# i n Pyt hon
df . sel ect ( "DEST_COUNTRY_NAME", "ORI GI N_COUNTRY_NAME") . show( 2)

- - i n SQL
SELECT DEST_COUNTRY_NAME, ORI GI N_COUNTRY_NAME FROM df Tabl e LI MI T 2 Giving
```

an output of:

```
+- - - - - - - - - - - - - - - - +- - - - - - - - - - - - - - - - - - + |DEST_COUNTRY_NAME|ORI
GI N_COUNTRY_NAME|
+- - - - - - - - - - - - - - - - +- - - - - - - - - - - - - - - - - - +
| Uni t ed St at es| Romani a|
| Uni t ed St at es| Croat i a|
+- - - - - - - - - - - - - - - - +- - - - - - - - - - - - - - - - - - +
```

As discussed in "Columns and Expressions", you can refer to columns in a number of different ways; all you need to keep in mind is that you can use them interchangeably:

```
// i n Scal a i mport org. apache. spark. sql . f unct i ons. { expr, col , col
umn} df . sel ect ( df . col ( "DEST_COUNTRY_NAME") , col (
"DEST_COUNTRY_NAME") , col umn( "DEST_COUNTRY_NAME") ,
  ' DEST_COUNTRY_NAME,
$"DEST_COUNTRY_NAME", expr(
"DEST_COUNTRY_NAME") ) . show(
2)

# i n Pyt hon
f rom pyspark. sql . f unct i ons i mport expr, col , col umn df . sel
ect ( expr( "DEST_COUNTRY_NAME") , col (
"DEST_COUNTRY_NAME") , col umn(
"DEST_COUNTRY_NAME") ) \
 . show( 2)
```

One common error is attempting to mix Col umn objects and strings. For example, the following code will result in a compiler error: df . sel ect (col ("DEST_COUNTRY_NAME") , "DEST_COUNTRY_NAME")

As we've seen thus far, expr is the most flexible reference that we can use. It can refer to a plain column or a string manipulation of a column. To illustrate, let's change the column name, and then change it back by using the AS keyword and then the al i as method on the column:

```
// i n Scal a
df . sel ect ( expr( "DEST_COUNTRY_NAME AS dest i nat i on") ) . show( 2)

# i n Pyt hon
df . sel ect ( expr( "DEST_COUNTRY_NAME AS dest i nat i on") ) . show( 2)

- - i n SQL
SELECT DEST_COUNTRY_NAME as dest i nat i on FROM df Tabl e LI MI T 2
```

This changes the column name to "destination." You can further manipulate the result of your expression as another expression:

```
// i n Scal a df . sel ect ( expr( "DEST_COUNTRY_NAME as dest i nat i on") . al i as(
"DEST_COUNTRY_NAME") ) . show( 2)

# i n Pyt hon
df . sel ect ( expr( "DEST_COUNTRY_NAME as dest i nat i on") . al i as( "DEST_COUNTRY_NAME") ) \ .
show( 2)
```

The preceding operation changes the column name back to its original name.

Because sel ect followed by a series of expr is such a common pattern, Spark has a shorthand for doing this efficiently: sel ect Expr. This is probably the most convenient interface for everyday use:

```
// in Scala df.selectExpr("DEST_COUNTRY_NAME as newColumnName",
"DEST_COUNTRY_NAME").show(2)

# in Python
df.selectExpr("DEST_COUNTRY_NAME as newColumnName", "DEST_COUNTRY_NAME").show(
2)
```

This opens up the true power of Spark. We can treat selectExpr as a simple way to build up complex expressions that create new DataFrames. In fact, we can add any valid non-aggregating SQL statement, and as long as the columns resolve, it will be valid! Here's a simple example that adds a new column withinCountry to our DataFrame that specifies whether the destination and origin are the same:

```
// in Scala df.select
Expr(
  "*", // include all original columns
  "(DEST_COUNTRY_NAME = ORIGIN_COUNTRY_NAME) as withinCountry")
.show(2)

# in Python df.
selectExpr(
 "*", # all original columns
 "(DEST_COUNTRY_NAME = ORIGIN_COUNTRY_NAME) as withinCountry") \
.show(2)

--in SQL
SELECT *, (DEST_COUNTRY_NAME = ORIGIN_COUNTRY_NAME) as withinCountry
FROM dfTableLI
MIT 2
```

Giving an output of:

```
+-----------------+-------------------+-----+-------------+|DEST_COUNTRY_NAME|ORI
GIN_COUNTRY_NAME|count|withinCountry|
+-----------------+-------------------+-----+-------------+
|  United States|          Romania|   15|      false|
|  United States|          Croatia|    1|      false|
+-----------------+-------------------+-----+-------------+
```

With select expression, we can also specify aggregations over the entire DataFrame by taking advantage of the functions that we have. These look just like what we have been showing so far:

```
// in Scala df.selectExpr("avg(count)", "count(distinct(DEST_COUNTRY_NAME))")
.show(2)

# in Python
```

```
df . sel ect Expr( "avg( count ) ", "count ( di st i nct ( DEST_COUNTRY_NAME) ) ") . show( 2)

- - i n SQL
SELECT avg( count ) , count ( di st i nct ( DEST_COUNTRY_NAME) ) FROM df Tabl e LI MI T 2
```

Giving an output of:

```
+- - - - - - - - - - +- - - - - - - - - - - - - - - - - - - - - - - - - - - -
+ | avg( count ) |count ( DI STI NCT DEST_COUNTRY_NAME)
|
+- - - - - - - - - - +- - - - - - - - - - - - - - - - - - - - - - - - - - - -
+ |1770. 765625|                  132| +- - - - - - - - - - +- - -
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - +
```

Converting to Spark Types (Literals)

Sometimes, we need to pass explicit values into Spark that are just a value (rather than a new column). This might be a constant value or something we'll need to compare to later on. The way we do this is through *literals*. This is basically a translation from a given programming language's literal value to one that Spark understands. Literals are expressions and you can use them in the same way:

```
// i n Scal a i mport org. apache. spark. sql . f unct i ons. l i
t df . sel ect ( expr( "*") , l i t ( 1) . as( "One") ) . show( 2)

# i n Pyt hon f rom pyspark. sql . f unct i ons i mport l i t df . sel
ect ( expr( "*") , l i t ( 1) . al i as( "One") ) . show( 2)
```

In SQL, literals are just the specific value:

```
- - i n SQL
SELECT *, 1 as One FROM df Tabl e LI MI T 2
```

Giving an output of:

```
+- - - - - - - - - - - - - - - - +- - - - - - - - - - - - - - - - - - +- - - - - +- - - + |DEST_COUNTRY_NAME|ORI
GI N_COUNTRY_NAME|count |One|
+- - - - - - - - - - - - - - - - +- - - - - - - - - - - - - - - - - - +- - - - - +- - - +
|   Uni t ed St at es|         Romani a|   15|  1|
|   Uni t ed St at es|         Croat i a|    1|  1|
+- - - - - - - - - - - - - - - - +- - - - - - - - - - - - - - - - - - +- - - - - +- - - +
```

This will come up when you might need to check whether a value is greater than some constant or other programmatically created variable.

Adding Columns

There's also a more formal way of adding a new column to a DataFrame, and that's by using the wi t hCol umn method on our DataFrame. For example, let's add a column that just adds the number one as a column:

```
// i n Scal a df . wi t hCol umn( "numberOne", l i t ( 1)
) . show( 2)

# i n Pyt hon df . wi t hCol umn( "numberOne", l i t (
1) ) . show( 2)

- - i n SQL
SELECT *, 1 as numberOne FROM df Tabl e LI MI T 2
```

Giving an output of:

```
+-----------------+-------------------+-----+---------+|DEST_COUNTRY_NAME|ORI
GI N_COUNTRY_NAME|count |numberOne|
+-----------------+-------------------+-----+---------+
|   Uni t ed St at es|         Romani a|   15|      1|
|   Uni t ed St at es|         Croat i a|    1|      1|
+-----------------+-------------------+-----+---------+
```

Let's do something a bit more interesting and make it an actual expression. In the next example, we'll set a Boolean flag for when the origin country is the same as the destination country:

```
// i n Scal a df . wi t hCol umn( "wi t hi nCount ry", expr( "ORI GI N_COUNTRY_NAME ==
DEST_COUNTRY_NAME") )  . show( 2)

# i n Pyt hon df . wi t hCol umn( "wi t hi nCount ry", expr( "ORI GI N_COUNTRY_NAME ==
DEST_COUNTRY_NAME") ) \  . show( 2)
```

Notice that the wi t hCol umn function takes two arguments: the column name and the expression that will create the value for that given row in the DataFrame. Interestingly, we can also rename a column this way. The SQL syntax is the same as we had previously, so we can omit it in this example: `df . wi t hCol umn( "Dest i nat i on", expr( "DEST_COUNTRY_NAME") ) . col umns`

Resulting in:

```
. . . DEST_COUNTRY_NAME, ORI GI N_COUNTRY_NAME, count , Dest i nat i on
```

Renaming Columns

Although we can rename a column in the manner that we just described, another alternative is to use the wi t hCol umnRenamed method. This will rename the column with the name of the string in the first argument to the string in the second argument:

```
// i n Scal a df . wi t hCol umnRenamed( "DEST_COUNTRY_NAME", "dest
") . col umns

# i n Pyt hon df . wi t hCol umnRenamed( "DEST_COUNTRY_NAME",
"dest ") . col umns

. . . dest , ORI GI N_COUNTRY_NAME, count
```

Reserved Characters and Keywords

One thing that you might come across is reserved characters like spaces or dashes in column names. Handling these means escaping column names appropriately. In Spark, we do this by using backtick (`) characters. Let's use wi t hCol umn, which you just learned about to create a column with reserved characters. We'll show two examples—in the one shown here, we don't need escape characters, but in the next one, we do:

```
// i n Scal a i mport org. apache. spark. sql . f unct i
ons. expr

val df Wi t hLongCol Name = df . wi t hCol umn(
 "Thi s Long Col umn- Name",  expr(
"ORI GI N_COUNTRY_NAME") )

# i n Pyt hon df Wi t hLongCol Name = df .
wi t hCol umn(     "Thi s Long Col umn-
Name",                  expr(    "ORI    GI
N_COUNTRY_NAME") )
```

We don't need escape characters here because the first argument to wi t hCol umn is just a string for the new column name. In this example, however, we need to use backticks because we're referencing a column in an expression:

```
// i n Scal a df Wi t hLongCol Name.
sel ect Expr(
  "` Thi s Long Col umn- Name` ",
  "` Thi s Long Col umn- Name` as ` new col ` ")
 . show( 2)

# i n Pyt hon
df Wi t hLongCol Name. sel ect Expr(
  "` Thi s Long Col umn- Name` ",
```

```
    "` Thi s Long Col umn- Name` as ` new col ` ") \
  . show( 2)

df Wi t hLongCol Name. creat eOrRepl aceTempVi ew( "df Tabl eLong")

- - i n SQL
SELECT ` Thi s Long Col umn- Name` , ` Thi s Long Col umn- Name` as ` new col ` FROM df
Tabl eLong LI MI T 2
```

We can refer to columns with reserved characters (and not escape them) if we're doing an explicit string-to-column reference, which is interpreted as a literal instead of an expression. We only need to escape expressions that use reserved characters or keywords. The following two examples both result in the same DataFrame:

```
// i n Scal a df Wi t hLongCol Name. sel ect ( col ( "Thi s Long Col umn- Name")
) . col umns

# i n Pyt hon
df Wi t hLongCol Name. sel ect ( expr( "` Thi s Long Col umn- Name` ") ) . col umns
```

Case Sensitivity

By default Spark is case insensitive; however, you can make Spark case sensitive by setting the configuration:

```
- - i n SQL
set spark. sql . caseSensi t i ve t rue
```

Removing Columns

Now that we've created this column, let's take a look at how we can remove columns from DataFrames. You likely already noticed that we can do this by using sel ect . However, there is also a dedicated method called drop: df . drop("ORI GI N_COUNTRY_NAME") . col umns

We can drop multiple columns by passing in multiple columns as arguments:

```
df Wi t hLongCol Name. drop( "ORI GI N_COUNTRY_NAME", "DEST_COUNTRY_NAME")
```

Changing a Column's Type (cast)

Sometimes, we might need to convert from one type to another; for example, if we have a set of

St ri ngType that should be integers. We can convert columns from one type to another by casting the column from one type to another. For instance, let's convert our count column from an integer to a type Long: df . wi t hCol umn("count 2", col ("count ") . cast ("l ong"))

```
- - i n SQL
SELECT *, cast ( count as l ong) AS count 2 FROM df Tabl e
```

Filtering Rows

To filter rows, we create an expression that evaluates to true or false. You then filter out the rows with an expression that is equal to false. The most common way to do this with DataFrames is to create either an expression as a String or build an expression by using a set of column manipulations. There are two methods to perform this operation: you can use where or f i l t er and they both will perform the same operation and accept the same argument types when used with DataFrames. We will stick to where because of its familiarity to SQL; however, f i l t er is valid as well.

> **NOTE**
>
> When using the Dataset API from either Scala or Java, f i l t er also accepts an arbitrary function that Spark will apply to each record in the Dataset. See Chapter 11 for more information.

The following filters are equivalent, and the results are the same in Scala and Python:

```
df . f i l t er( col ( "count ") < 2) . show( 2) df .
where( "count < 2") . show( 2)

- - i n SQL
SELECT * FROM df Tabl e WHERE count < 2 LI MI T 2
```

Giving an output of:

```
+- - - - - - - - - - - - - - - - +- - - - - - - - - - - - - - - - - - +- - - - - + |DEST_COUNTRY_NAME|ORI
GI N_COUNTRY_NAME|count |
+- - - - - - - - - - - - - - - - +- - - - - - - - - - - - - - - - - - +- - - - - +
|   Uni t ed St at es|          Croat i a|    1|
|   Uni t ed St at es|        Si ngapore|    1|
+- - - - - - - - - - - - - - - - +- - - - - - - - - - - - - - - - - - +- - - - - +
```

Instinctually, you might want to put multiple filters into the same expression. Although this is possible, it is not always useful, because Spark automatically performs all filtering operations at the same time regardless of the filter ordering. This means that if you want to specify multiple AND filters, just chain them sequentially and let Spark handle the rest:

```
// in Scala df.where(col("count") < 2).where(col("ORIGIN_COUNTRY_NAME") =!=
"Croatia").show(2)

# in Python df.where(col("count") < 2).where(col("ORIGIN_COUNTRY_NAME") !=
"Croatia") \  .show(2)

--in SQL
SELECT * FROM dfTable WHERE count < 2 AND ORIGIN_COUNTRY_NAME != "Croatia" LIMI
T 2
```

Giving an output of:

```
+-----------------+-------------------+-----+|DEST_COUNTRY_NAME|ORI
GIN_COUNTRY_NAME|count|
+-----------------+-------------------+-----+
|  United States|        Singapore|    1|
|       Moldova|     United States|    1|
+-----------------+-------------------+-----+
```

Getting Unique Rows

A very common use case is to extract the unique or distinct values in a DataFrame. These values can be in one or more columns. The way we do this is by using the distinct method on a DataFrame, which allows us to deduplicate any rows that are in that DataFrame. For instance, let's get the unique origins in our dataset. This, of course, is a transformation that will return a new DataFrame with only unique rows:

```
// in Scala df.select("ORIGIN_COUNTRY_NAME", "DEST_COUNTRY_NAME").distinct
().count()

# in Python
df.select("ORIGIN_COUNTRY_NAME", "DEST_COUNTRY_NAME").distinct().count()

--in SQL
SELECT COUNT(DISTINCT(ORIGIN_COUNTRY_NAME, DEST_COUNTRY_NAME)) FROM dfTable
```

Results in 256.

```
// in Scala df.select("ORIGIN_COUNTRY_NAME").distinct
().count()

# in Python
df.select("ORIGIN_COUNTRY_NAME").distinct().count()

--in SQL
SELECT COUNT(DISTINCT ORIGIN_COUNTRY_NAME) FROM dfTable
```

Results in 125.

Random Samples

Sometimes, you might just want to sample some random records from your DataFrame. You can do this by using the sampl e method on a DataFrame, which makes it possible for you to specify a fraction of rows to extract from a DataFrame and whether you'd like to sample with or without replacement:

```
val seed = 5 val wi t hRepl acement
= f al se val f ract i on = 0. 5
df . sampl e( wi t hRepl acement , f ract i on, seed) . count ( )

# i n Pyt hon seed = 5 wi t
hRepl acement = Fal se f ract
i on = 0. 5
df . sampl e( wi t hRepl acement , f ract i on, seed) . count ( )
```

Giving an output of 126.

Random Splits

Random splits can be helpful when you need to break up your DataFrame into a random "splits" of the original DataFrame. This is often used with machine learning algorithms to create training, validation, and test sets. In this next example, we'll split our DataFrame into two different DataFrames by setting the weights by which we will split the DataFrame (these are the arguments to the function). Because this method is designed to be randomized, we will also specify a seed (just replace seed with a number of your choosing in the code block). It's important to note that if you don't specify a proportion for each DataFrame that adds up to one, they will be normalized so that they do:

```
// i n Scal a val dat aFrames = df . randomSpl i t ( Array( 0. 25, 0. 75) ,
seed) dat aFrames( 0) . count ( ) > dat aFrames( 1) . count ( ) // Fal se

# i n Pyt hon dat aFrames = df . randomSpl i t ( [ 0. 25, 0. 75] , seed)
dat aFrames[ 0] . count ( ) > dat aFrames[ 1] . count ( ) # Fal se
```

Concatenating and Appending Rows (Union)

As you learned in the previous section, DataFrames are immutable. This means users cannot append to DataFrames because that would be changing it. To append to a DataFrame, you must *union* the original DataFrame along with the new DataFrame. This just concatenates the two DataFramess. To union two DataFrames, you must be sure that they have the same schema and number of columns; otherwise, the union will fail.

WARNING

Unions are currently performed based on location, not on the schema. This means that columns will not automatically line up the way you think they might.

```
// i n Scal a i mport org. apache. spark.
sql . Row val schema = df . schema val
newRows = Seq(
 Row( "New Count ry", "Ot her Count ry", 5L) ,
 Row( "New Count ry 2", "Ot her Count ry 3", 1L)
)
val paral l el i zedRows = spark. sparkCont ext . paral l el i ze( newRows) val
newDF = spark. creat eDat aFrame( paral l el i zedRows, schema) df . uni on(
newDF)
 . where( "count = 1")
 . where( $"ORI GI N_COUNTRY_NAME" =! = "Uni t ed St at es")
 . show( ) // get al l of t hem and we' l l see our new rows at t he end
```

In Scala, you must use the =! = operator so that you don't just compare the unevaluated column expression to a string but instead to the evaluated one:

```
# i n Pyt hon f rom pyspark. sql i
mport Row schema = df . schema
newRows = [
 Row( "New Count ry", "Ot her Count ry", 5L) ,
 Row( "New Count ry 2", "Ot her Count ry 3", 1L)
]
paral l el i zedRows = spark. sparkCont ext . paral l el i ze( newRows) newDF =
spark. creat eDat aFrame( paral l el i zedRows, schema)

# i n Pyt hon df . uni
on( newDF) \
 . where( "count = 1") \
 . where( col ( "ORI GI N_COUNTRY_NAME") ! = "Uni t ed St at es") \  .
show( )
```

Giving the output of:

```
+----------------+-------------------+-----+ |DEST_COUNTRY_NAME|ORI
GI N_COUNTRY_NAME|count |
+-----------------+-------------------+-----+
|   Uni t ed St at es|         Croat i a|    1| . . .
|   Uni t ed St at es|         Nami bi a|    1|
|   New Count ry 2|    Ot her Count ry 3|    1|
+-----------------+-------------------+-----+
```

As expected, you'll need to use this new DataFrame reference in order to refer to the DataFrame with the newly appended rows. A common way to do this is to make the DataFrame into a view or register it as a table so that you can reference it more dynamically in your code.

Sorting Rows

When we sort the values in a DataFrame, we always want to sort with either the largest or smallest values at the top of a DataFrame. There are two equivalent operations to do this sort and orderBy that work the exact same way. They accept both column expressions and strings as well as multiple columns. The default is to sort in ascending order:

```
// in Scala df.sort("count").show(5) df.orderBy("count",
"DEST_COUNTRY_NAME").show(5) df.orderBy(col("count"),col(
"DEST_COUNTRY_NAME")).show(5)

# in Python df.sort("count").show(5) df.orderBy("count",
"DEST_COUNTRY_NAME").show(5) df.orderBy(col("count"),col(
"DEST_COUNTRY_NAME")).show(5)
```

To more explicitly specify sort direction, you need to use the asc and desc functions if operating on a column. These allow you to specify the order in which a given column should be sorted:

```
// in Scala import org.apache.spark.sql.functions.{desc,
asc} df.orderBy(expr("count desc")).show(2)
df.orderBy(desc("count"),asc("DEST_COUNTRY_NAME")).show(2)

# in Python from pyspark.sql.functions import
desc, asc df.orderBy(expr("count desc")).show(2)
df.orderBy(col("count").desc(),col("DEST_COUNTRY_NAME").asc()).show(2)

--in SQL
SELECT * FROM dfTable ORDER BY count DESC, DEST_COUNTRY_NAME ASC LIMIT 2
```

An advanced tip is to use asc_nulls_first, desc_nulls_first, asc_nulls_last, or desc_nulls_last to specify where you would like your null values to appear in an ordered DataFrame.

For optimization purposes, it's sometimes advisable to sort within each partition before another set of transformations. You can use the sortWithinPartitions method to do this:

```
// in Scala spark.read.format("json").load("/data/flight-data/json/*-summary.
json")  .sortWithinPartitions("count")

# in Python spark.read.format("json").load("/data/flight-data/json/*-summary.
json")\  .sortWithinPartitions("count")
```

We will discuss this more when we look at tuning and optimization in Part III.

Limit

Oftentimes, you might want to restrict what you extract from a DataFrame; for example, you might want just the top ten of some DataFrame. You can do this by using the l i mi t method:

```
// i n Scal a df . l i mi t
( 5) . show( )

# i n Pyt hon df . l i mi
t ( 5) . show( )

- - i n SQL
SELECT * FROM df Tabl e LI MI T 6

// i n Scal a df . orderBy( expr( "count desc") ) . l i mi t ( 6)
. show( )

# i n Pyt hon df . orderBy( expr( "count desc") ) . l i mi t ( 6)
. show( )

- - i n SQL
SELECT * FROM df Tabl e ORDER BY count desc LI MI T 6
```

Repartition and Coalesce

Another important optimization opportunity is to partition the data according to some frequently filtered columns, which control the physical layout of data across the cluster including the partitioning scheme and the number of partitions.

Repartition will incur a full shuffle of the data, regardless of whether one is necessary. This means that you should typically only repartition when the future number of partitions is greater than your current number of partitions or when you are looking to partition by a set of columns:

```
// i n Scal a df . rdd. get NumPart i t
i ons // 1

# i n Pyt hon df . rdd. get NumPart i t
i ons( ) # 1 // i n Scal a df . repart i t i
on( 5)

# i n Pyt hon df .
repart i t i on( 5)
```

If you know that you're going to be filtering by a certain column often, it can be worth repartitioning based on that column:

```
// in Scala df.repartition(col("DEST_COUNTRY_NAME"))

# in Python
df.repartition(col("DEST_COUNTRY_NAME"))
```

You can optionally specify the number of partitions you would like, too:

```
// in Scala df.repartition(5, col("DEST_COUNTRY_NAME"))

# in Python
df.repartition(5, col("DEST_COUNTRY_NAME"))
```

Coalesce, on the other hand, will not incur a full shuffle and will try to combine partitions. This operation will shuffle your data into five partitions based on the destination country name, and then coalesce them (without a full shuffle):

```
// in Scala df.repartition(5, col("DEST_COUNTRY_NAME")).coal
esce(2)

# in Python
df.repartition(5, col("DEST_COUNTRY_NAME")).coalesce(2)
```

Collecting Rows to the Driver

As discussed in previous chapters, Spark maintains the state of the cluster in the driver. There are times when you'll want to collect some of your data to the driver in order to manipulate it on your local machine.

Thus far, we did not explicitly define this operation. However, we used several different methods for doing so that are effectively all the same. collect gets all data from the entire DataFrame, take selects the first *N* rows, and show prints out a number of rows nicely.

```
// in Scala val collectDF = df.limit(10) collectDF.take(5) //
take works with an Integer count collectDF.show() // this pri
nts it out nicely
collectDF.show(5, false) collect
ectDF.collect()

# in Python collectDF = df.limit(10) collectDF.take(5) # t
ake works with an Integer count collectDF.show() # this pri
nts it out nicely collectDF.show(5, False) collectDF.collect
()
```

There's an additional way of collecting rows to the driver in order to iterate over the entire dataset. The method t oLocal I t erat or collects partitions to the driver as an iterator. This method allows you to iterate over the entire dataset partition-by-partition in a serial manner:

```
col l ect DF. t oLocal I t erat or( )
```

> **WARNING**
>
> Any collection of data to the driver can be a very expensive operation! If you have a large dataset and call col l ect , you can crash the driver. If you use t oLocal I t erat or and have very large partitions, you can easily crash the driver node and lose the state of your application. This is also expensive because we can operate on a one-by-one basis, instead of running computation in parallel.

Conclusion

This chapter covered basic operations on DataFrames. You learned the simple concepts and tools that you will need to be successful with Spark DataFrames. Chapter 6 covers in much greater detail all of the different ways in which you can manipulate the data in those DataFrames.

Chapter 6. Working with Different Types of Data

Chapter 5 presented basic DataFrame concepts and abstractions. This chapter covers building expressions, which are the bread and butter of Spark's structured operations. We also review working with a variety of different kinds of data, including the following:

- Booleans
- Numbers
- Strings
- Dates and timestamps
- Handling null
- Complex types User-
- defined functions

Where to Look for APIs

Before we begin, it's worth explaining where you as a user should look for transformations. Spark is a growing project, and any book (including this one) is a snapshot in time. One of our priorities in this book is to teach where, as of this writing, you should look to find functions to transform your data. Following are the key places to look:

Dat aFrame (Dat aset) Methods

> This is actually a bit of a trick because a DataFrame is just a Dataset of Row types, so you'll actually end up looking at the Dat aset methods, which are available at this link.

Dat aset submodules like Dat aFrameSt at Funct i ons and Dat aFrameNaFunct i ons have more methods that solve specific sets of problems. Dat aFrameSt at Funct i ons, for example, holds a variety of statistically related functions, whereas Dat aFrameNaFunct i ons refers to functions that are relevant when working with null data.

Col umn Methods

> These were introduced for the most part in Chapter 5. They hold a variety of general columnrelated methods like al i as or cont ai ns. You can find the API Reference for Column methods here.

org. apache. spark. sql . f unct i ons contains a variety of functions for a range of different data types. Often, you'll see the entire package imported because they are used so frequently. You can find SQL and DataFrame functions here.

Now this may feel a bit overwhelming but have no fear, the majority of these functions are ones that you will find in SQL and analytics systems. All of these tools exist to achieve one purpose, to transform rows of data in one format or structure to another. This might create more rows or reduce the number of rows available. To begin, let's read in the Dat aFrame that we'll be using for this analysis:

```
// i n Scal a val df = spark. read. f ormat (
"csv")   . opt i on( "header", "t rue")
  . opt i on( "i nf erSchema", "t rue")
  . l oad( "/dat a/ret ai l - dat a/by- day/2010- 12- 01. csv") df .
pri nt Schema( ) df . creat eOrRepl aceTempVi ew( "df Tabl e")

# i n Pyt hon df = spark. read. f ormat
( "csv") \   . opt i on( "header", "t rue")
\
  . opt i on( "i nf erSchema", "t rue") \
  . l oad( "/dat a/ret ai l - dat a/by- day/2010- 12- 01. csv") df . pri
nt Schema( )
df . creat eOrRepl aceTempVi ew( "df Tabl e")
```

Here's the result of the schema and a small sample of the data:

```
root
 |-- InvoiceNo: string (nullable = true)
 |-- StockCode: string (nullable = true)
 |-- Description: string (nullable = true)
 |-- Quantity: integer (nullable = true)
 |-- InvoiceDate: timestamp (nullable = true)
 |-- UnitPrice: double (nullable = true)
 |-- CustomerID: double (nullable = true)
 |-- Country: string (nullable = true)

+---------+---------+--------------------+--------+-------------------+----...
|InvoiceNo|StockCode|         Description|Quantity|        InvoiceDate|Unit...
+---------+---------+--------------------+--------+-------------------+----...
|   536365|   85123A|WHITE HANGING HEA...|       6|2010-12-01 08:26:00|   ...|
536365|    71053| WHITE METAL LANTERN|       6|2010-12-01 08:26:00|   .....
|   536367|    21755|LOVE BUILDING BLO...|       3|2010-12-01 08:34:00|   ...
|   536367|    21777|RECIPE BOX WITH M...|       4|2010-12-01 08:34:00|   ...
+---------+---------+--------------------+--------+-------------------+----...
```

Converting to Spark Types

One thing you'll see us do throughout this chapter is convert native types to Spark types. We do this by using the first function that we introduce here, the lit function. This function converts a type in another language to its correspnding Spark representation. Here's how we can convert a couple of different kinds of Scala and Python values to their respective Spark types:

```
// in Scala
import org.apache.spark.sql.functions.lit
df.select(lit(5),lit("five"),lit(5.0))
```

```
# in Python
from pyspark.sql.functions import lit
df.select(lit(5),lit("five"),lit(5.0))
```

There's no equivalent function necessary in SQL, so we can use the values directly:

```
-- in SQL
SELECT 5, "five", 5.0
```

Working with Booleans

Booleans are essential when it comes to data analysis because they are the foundation for all filtering. Boolean statements consist of four elements: *and*, *or*, *true*, and *false*. We use these simple structures to build logical statements that evaluate to either *true* or *false*. These statements are often used as conditional requirements for when a row of data must either pass the test (evaluate to true) or else it will be filtered out.

Let's use our retail dataset to explore working with Booleans. We can specify equality as well as less-than or greater-than:

```
// i n Scal a i mport org. apache. spark. sql . f unct i
ons. col df . where( col ( "I nvoi ceNo") . equal To(
536365) )
  . sel ect ( "I nvoi ceNo", "Descri pt i on")
  . show( 5, f al se)
```

WARNING

Scala has some particular semantics regarding the use of == and ===. In Spark, if you want to filter by equality you should use === (equal) or =! = (not equal). You can also use the not function and the equal To method.

```
// i n Scal a i mport org. apache. spark. sql . f unct i
ons. col df . where( col ( "I nvoi ceNo") === 536365)
. sel ect ( "I nvoi ceNo", "Descri pt i on")
  . show( 5, f al se)
```

Python keeps a more conventional notation:

```
# i n Pyt hon f rom pyspark. sql . f unct i ons i
mport col df . where( col ( "I nvoi ceNo") ! =
536365) \
  . sel ect ( "I nvoi ceNo", "Descri pt i on") \
  . show( 5, Fal se)
```

```
+- - - - - - - - -+- - - - - - - - - - - - - - - - - - - - - - - - - - - +
|I nvoi ceNo|Descri pt i on                |
+- - - - - - - - -+- - - - - - - - - - - - - - - - - - - - - - - - - - - +
|536366   |HAND WARMER UNI ON J ACK      | . . .
|536367   |POPPY' S PLAYHOUSE KI TCHEN    |
+- - - - - - - - -+- - - - - - - - - - - - - - - - - - - - - - - - - - - +
```

Another option—and probably the cleanest—is to specify the predicate as an expression in a string. This is valid for Python or Scala. Note that this also gives you access to another way of expressing "does not equal":

```
df . where( "I nvoi ceNo = 536365")
  . show( 5, f al se)

df . where( "I nvoi ceNo <> 536365")
  . show( 5, f al se)
```

We mentioned that you can specify Boolean expressions with multiple parts when you use and or or. In Spark, you should always chain together and filters as a sequential filter.

The reason for this is that even if Boolean statements are expressed serially (one after the other), Spark will flatten all of these filters into one statement and perform the filter at the same time, creating the and statement for us. Although you can specify your statements explicitly by using and if you like, they're often easier to understand and to read if you specify them serially. or statements need to be specified in the same statement:

```
// in Scala val priceFilter = col("UnitPrice") > 600 val descripFilter = col("Descript
ion").contains("POSTAGE") df.where(col("StockCode").isin("DOT")).where(priceFil
ter.or(descripFilter)) .show()

# in Python from pyspark.sql.functions import instr priceFil
ter = col("UnitPrice") > 600 descripFilter = instr(df.Descript
ion, "POSTAGE") >= 1
df.where(df.StockCode.isin("DOT")).where(priceFilter | descripFilter).show()

--in SQL
SELECT * FROM dfTable WHERE StockCode in ("DOT") AND(UnitPrice > 600 OR
  instr(Description, "POSTAGE") >= 1)
```

```
+---------+---------+--------------+--------+-------------------+---------+...
|InvoiceNo|StockCode|   Description|Quantity|    InvoiceDate|UnitPrice|...
+---------+---------+--------------+--------+-------------------+---------+...
|   536544|      DOT|DOTCOM POSTAGE|       1|2010-12-01 14:32:00|   569.77|...
|   536592|      DOT|DOTCOM POSTAGE|       1|2010-12-01 17:06:00|   607.49|...
+---------+---------+--------------+--------+-------------------+---------+...
```

Boolean expressions are not just reserved to filters. To filter a DataFrame, you can also just specify a Boolean column:

```
// in Scala val DOTCodeFilter = col("StockCode") === "DOT" val priceFilter = col("Unit
Price") > 600 val descripFilter = col("Description").contains("POSTAGE") df.withCol
umn("isExpensive", DOTCodeFilter.and(priceFilter.or(descripFilter)))
  .where("isExpensive")
  .select("unitPrice", "isExpensive").show(5)

# in Python
from pyspark.sql.functions import instr DOTCodeFilter = col("StockCode") == "DOT" pri
ceFilter = col("UnitPrice") > 600 descripFilter = instr(col("Description"), "POSTAGE")
>= 1 df.withColumn("isExpensive", DOTCodeFilter & (priceFilter | descripFilter)) \
  .where("isExpensive") \
  .select("unitPrice", "isExpensive").show(5)

--in SQL
SELECT UnitPrice, (StockCode = 'DOT' AND
  (UnitPrice > 600 OR instr(Description, "POSTAGE") >= 1)) as isExpensive
FROM dfTable
WHERE (StockCode = 'DOT' AND
      (UnitPrice > 600 OR instr(Description, "POSTAGE") >= 1))
```

Notice how we did not need to specify our filter as an expression and how we could use a column name without any extra work.

If you're coming from a SQL background, all of these statements should seem quite familiar. Indeed, all of them can be expressed as a where clause. In fact, it's often easier to just express filters as SQL statements than using the programmatic Dat aFrame interface and Spark SQL allows us to do this without paying any performance penalty. For example, the following two statements are equivalent:

```
// i n Scal a i mport org. apache. spark. sql . f unct i ons. { expr, not , col } df
. wi t hCol umn( "i sExpensi ve", not ( col ( "Uni t Pri ce") . l eq( 250) ) )
  . f i l t er( "i sExpensi ve")
  . sel ect ( "Descri pt i on", "Uni t Pri ce") . show( 5)
df . wi t hCol umn( "i sExpensi ve", expr( "NOT Uni t Pri ce <= 250") )
  . f i l t er( "i sExpensi ve")
  . sel ect ( "Descri pt i on", "Uni t Pri ce") . show( 5)
```

Here's our state definition:

```
# i n Pyt hon f rom pyspark. sql . f unct i ons i mport expr df . wi t hCol umn(
"i sExpensi ve", expr( "NOT Uni t Pri ce <= 250") ) \
  . where( "i sExpensi ve") \
  . sel ect ( "Descri pt i on", "Uni t Pri ce") . show( 5)
```

> **WARNING**
>
> One "gotcha" that can come up is if you're working with null data when creating Boolean expressions. If there is a null in your data, you'll need to treat things a bit differently. Here's how you can ensure that you perform a null-safe equivalence test:
>
> ```
> df . where( col ( "Descri pt i on") . eqNul l Saf e( "hel l o") ) . show( )
> ```

Although not currently available (Spark 2.2), I S [NOT] DI STI NCT FROM will be coming in Spark 2.3 to do the same thing in SQL.

Working with Numbers

When working with big data, the second most common task you will do after filtering things is counting things. For the most part, we simply need to express our computation, and that should be valid assuming that we're working with numerical data types.

To fabricate a contrived example, let's imagine that we found out that we mis-recorded the quantity in our retail dataset and the true quantity is equal to (the current quantity * the unit price)^2 + 5. This will introduce our first numerical function as well as the pow function that raises a column to the expressed power:

```
// in Scala import org.apache.spark.sql.functions.{expr,
pow}
val fabricatedQuantity = pow(col("Quantity") * col("UnitPrice"), 2) + 5 df.select(expr(
"CustomerId"), fabricatedQuantity.alias("real Quantity")).show(2)

# in Python from pyspark.sql.functions import expr, pow fabricatedQuantity = pow(col(
"Quantity") * col("UnitPrice"), 2) + 5 df.select(expr("CustomerId"), fabricatedQuantity.
alias("real Quantity")).show(2)

+----------+------------------+
|CustomerId|     real Quantity|
+----------+------------------+
|   17850.0|239.08999999999997|
|   17850.0|        418.7156|
+----------+------------------+
```

Notice that we were able to multiply our columns together because they were both numerical. Naturally we can add and subtract as necessary, as well. In fact, we can do all of this as a SQL expression, as well:

```
// in Scala df.sel
ect Expr(
 "CustomerId",
 "(POWER((Quantity * UnitPrice), 2.0) + 5) as real Quantity").show(2)

# in Python df.
select Expr(
 "CustomerId",
 "(POWER((Quantity * UnitPrice), 2.0) + 5) as real Quantity").show(2)

--in SQL
SELECT customerId, (POWER((Quantity * UnitPrice), 2.0) + 5) as real Quantity FROM dfTabl
e
```

Another common numerical task is rounding. If you'd like to just round to a whole number, oftentimes you can cast the value to an integer and that will work just fine. However, Spark also has more detailed functions for performing this explicitly and to a certain level of precision. In the following example, we round to one decimal place:

```
// in Scala import org.apache.spark.sql.functions.{round,
bround}
df.select(round(col("UnitPrice"), 1).alias("rounded"), col("UnitPrice")).show(5)
```

By default, the round function rounds up if you're exactly in between two numbers. You can round down by using the bround:

```
// in Scala import org.apache.spark.sql.functions.lit df.select
(round(lit("2.5")),bround(lit("2.5"))).show(2)

# in Python from pyspark.sql.functions import lit,round,bround
df.select(round(lit("2.5")),bround(lit("2.5"))).show(2)

--in SQL
SELECT round(2.5),bround(2.5)
+-------------+--------------+|round(
2.5,0)|bround(2.5,0)|
+-------------+--------------+
|          3.0|           2.0|
|          3.0|           2.0|
+-------------+--------------+
```

Another numerical task is to compute the correlation of two columns. For example, we can see the Pearson correlation coefficient for two columns to see if cheaper things are typically bought in greater quantities. We can do this through a function as well as through the DataFrame statistic methods:

```
// in Scala import org.apache.spark.sql.functions.{
corr}df.stat.corr("Quantity","UnitPrice")df.select
(corr("Quantity","UnitPrice")).show()

# in Python from pyspark.sql.functions import corr df
.stat.corr("Quantity","UnitPrice")df.select(corr(
"Quantity","UnitPrice")).show()

--in SQL
SELECT corr(Quantity,UnitPrice)FROM dfTable

+-------------------------+|corr(
Quantity,UnitPrice)|
+-------------------------+
|    -0.04112314436835551|
+-------------------------+
```

Another common task is to compute summary statistics for a column or set of columns. We can use the describe method to achieve exactly this. This will take all numeric columns and calculate the count, mean, standard deviation, min, and max. You should use this primarily for viewing in the console because the schema might change in the future:

```
// in Scala df.describe(
).show()

# in Python df.describ
e().show()
```

```
+-------+-----------------+-----------------+-----------------+ |summary|
Quant i t y|       Uni t Pri ce|       Cust omerI D|
+-------+-----------------+-----------------+-----------------+
|  count |           3108|           3108|           1968|
|   mean| 8. 627413127413128| 4. 151946589446603|15661. 388719512195|
| st ddev|26. 371821677029203|15. 638659854603892|1854. 4496996893627|
|    mi n|            - 24|            0. 0|         12431. 0|
|    max|            600|         607. 49|         18229. 0|
+-------+-----------------+-----------------+-----------------+
```

If you need these exact numbers, you can also perform this as an aggregation yourself by importing the functions and applying them to the columns that you need:

```
// i n Scal a i mport org. apache. spark. sql . f unct i ons. { count , mean, st ddev_pop, mi n,
max}

# i n Pyt hon
f rom pyspark. sql . f unct i ons i mport count , mean, st ddev_pop, mi n, max
```

There are a number of statistical functions available in the StatFunctions Package (accessible using st at as we see in the code block below). These are DataFrame methods that you can use to calculate a variety of different things. For instance, you can calculate either exact or approximate quantiles of your data using the approxQuant i l e method:

```
// i n Scal a val col Name = "Uni t Pri ce" val quant i l eProbs = Array( 0. 5) val rel Error
= 0. 05 df . st at . approxQuant i l e( "Uni t Pri ce", quant i l eProbs, rel Error) // 2. 51

# i n Pyt hon col Name =
"Uni t Pri ce" quant i l
eProbs = [ 0. 5] rel Error =
0. 05
df . st at . approxQuant i l e( "Uni t Pri ce", quant i l eProbs, rel Error) # 2. 51
```

You also can use this to see a cross-tabulation or frequent item pairs (be careful, this output will be large and is omitted for this reason):

```
// i n Scal a df . st at . crosst ab( "St ockCode", "Quant i t y") .
show( )

# i n Pyt hon df . st at . crosst ab( "St ockCode", "Quant i t y")
. show( )

// i n Scal a df . st at . f reqI t ems( Seq( "St ockCode", "Quant i t y") )
. show( )

# i n Pyt hon
df . st at . f reqI t ems( [ "St ockCode", "Quant i t y"] ) . show( )
```

As a last note, we can also add a unique ID to each row by using the function monot oni cal l y_i ncreasi ng_i d. This function generates a unique value for each row, starting with 0:

```
// i n Scal a i mport org. apache. spark. sql . f unct i ons. monot oni cal l y_i ncreasi
ng_i d df . sel ect ( monot oni cal l y_i ncreasi ng_i d( ) ) . show( 2)

# i n Pyt hon f rom pyspark. sql . f unct i ons i mport monot oni cal l y_i ncreasi
ng_i d df . sel ect ( monot oni cal l y_i ncreasi ng_i d( ) ) . show( 2)
```

There are functions added with every release, so check the documentation for more methods. For instance, there are some random data generation tools (e.g., rand( ) , randn( )) with which you can randomly generate data; however, there are potential determinism issues when doing so. (You can find discussions about these challenges on the Spark mailing list.) There are also a number of more advanced tasks like bloom filtering and sketching algorithms available in the stat package that we mentioned (and linked to) at the beginning of this chapter. Be sure to search the API documentation for more information and functions.

Working with Strings

String manipulation shows up in nearly every data flow, and it's worth explaining what you can do with strings. You might be manipulating log files performing regular expression extraction or substitution, or checking for simple string existence, or making all strings uppercase or lowercase.

Let's begin with the last task because it's the most straightforward. The i ni t cap function will capitalize every word in a given string when that word is separated from another by a space.

```
// i n Scal a i mport org. apache. spark. sql . f unct i ons. { i ni t cap}
df . sel ect ( i ni t cap( col ( "Descri pt i on") ) ) . show( 2, f al se)

# i n Pyt hon f rom pyspark. sql . f unct i ons i mport i ni t
cap df . sel ect ( i ni t cap( col ( "Descri pt i on") ) ) . show(
)

- - i n SQL
SELECT i ni t cap( Descri pt i on) FROM df Tabl e

+- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - +
|i ni t cap( Descri pt i on)           |
+- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - +
|Whi t e Hangi ng Heart T- l i ght Hol der|
|Whi t e Met al Lant ern           |
+- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - +
```

As just mentioned, you can cast strings in uppercase and lowercase, as well:

```
// in Scala
import org.apache.spark.sql.functions.{lower, upper}
df.select(col("Description"),
  lower(col("Description")),
  upper(lower(col("Description"))))).show(2)

# in Python
from pyspark.sql.functions import lower,
upper
df.select(col("Description"),
  lower(col("Description")),
  upper(lower(col("Description")))
)).show(2)

-- in SQL
SELECT Description, lower(Description), Upper(lower(Description)) FROM dfTable

+--------------------+--------------------+-------------------------
+|         Description|  lower(Description)|upper(lower(Description))|
+--------------------+--------------------+-------------------------+
|WHITE HANGING HEA...|white hanging hea...|     WHITE HANGING HEA...|
| WHITE METAL LANTERN| white metal lantern|      WHITE METAL LANTERN|
+--------------------+--------------------+-------------------------+
```

Another trivial task is adding or removing spaces around a string. You can do this by using lpad, ltrim, rpad and rtrim, trim:

```
// in Scala import org.apache.spark.sql.functions.{lit, ltrim, rtrim, rpad, lpad, trim}
df.select(    ltrim(lit("    HELLO    ")).as("ltrim"),    rtrim(lit("    HELLO    ")).as(
"rtrim"),    trim(lit("    HELLO    ")).as("trim"),    lpad(lit("HELLO"), 3, " ").as("lp")
,    rpad(lit("HELLO"), 10, " ").as("rp")).show(2)

# in Python from pyspark.sql.functions import lit, ltrim, rtrim, rpad, lpad, trim
df.select(    ltrim(lit("    HELLO    ")).alias("ltrim"),    rtrim(lit("    HELLO
")).alias("rtrim"),    trim(lit("    HELLO    ")).alias("trim"),    lpad(lit(
"HELLO"), 3, " ").alias("lp"),    rpad(lit("HELLO"), 10, " ").alias("rp")).show(2)

-- in SQL SELECT    ltrim('
HELLLOOOO    '),    rtrim('
HELLLOOOO    '),    trim('
HELLLOOOO '), lpad('HELLOOOO
', 3, ''), rpad('HELLOOOO ', 10,
'')
FROM dfTable
+---------+---------+-----+---+----------+
|  ltrim|   rtrim|trim|lp|        rp|
+---------+---------+-----+---+----------+
|HELLO    |    HELLO|HELLO| HE|HELLO     |
|HELLO    |    HELLO|HELLO| HE|HELLO     |
+---------+---------+-----+---+----------+
```

Note that if l pad or rpad takes a number less than the length of the string, it will always remove values from the right side of the string.

Regular Expressions

Probably one of the most frequently performed tasks is searching for the existence of one string in another or replacing all mentions of a string with another value. This is often done with a tool called *regular expressions* that exists in many programming languages. Regular expressions give the user an ability to specify a set of rules to use to either extract values from a string or replace them with some other values.

Spark takes advantage of the complete power of Java regular expressions. The Java regular expression syntax departs slightly from other programming languages, so it is worth reviewing before putting anything into production. There are two key functions in Spark that you'll need in order to perform regular expression tasks: regexp_ext ract and regexp_repl ace. These functions extract values and replace values, respectively.

Let's explore how to use the regexp_repl ace function to replace substitute color names in our description column:

```
// i n Scal a i mport org. apache. spark. sql . f unct i ons. regexp_repl ace val si mpl
eCol ors = Seq( "bl ack", "whi t e", "red", "green", "bl ue") val regexSt ri ng = si
mpl eCol ors. map( _. t oUpperCase) . mkSt ri ng( "|")
// t he | si gni f i es ` OR` i n regul ar expressi on synt ax df . sel ect (   regexp_repl ace( col ( "Descri pt
i on") , regexSt ri ng, "COLOR") . al i as( "col or_cl ean") ,   col ( "Descri pt i on") ) . show( 2)

# i  n  Pyt  hon  f  rom  pyspark.  sql  .  f  unct  i  ons  i  mport
regexp_repl   ace   regex_st   ri   ng   =   "BLACK|WHI
TE|RED|GREEN|BLUE" df . sel ect (
 regexp_repl ace( col ( "Descri pt i on") , regex_st ri ng, "COLOR") . al i as( "col or_cl ean") ,   col ( "Descri
pt i on") ) . show( 2)

- - i n SQL SELECT   regexp_repl ace( Descri pt i on, ' BLACK|WHI TE|RED|GREEN|BLUE' , '
COLOR' ) as   col or_cl ean, Descri pt i on
FROM df Tabl e
+- - - - - - - - - - - - - - - - - - - +- - - - - - - - - - - - - - - - - - -
+ |       col or_cl ean|       Descri pt i on|
+- - - - - - - - - - - - - - - - - - - +- - - - - - - - - - - - - - - - - - - +
|COLOR HANGI NG HEA. . . |WHI TE HANGI NG HEA. . . |
| COLOR METAL LANTERN| WHI TE METAL LANTERN|
+- - - - - - - - - - - - - - - - - - - +- - - - - - - - - - - - - - - - - - - +
```

Another task might be to replace given characters with other characters. Building this as a regular expression could be tedious, so Spark also provides the translate function to replace these values. This is done at the character level and will replace all instances of a character with the indexed character in the replacement string:

```
// in Scala import org.apache.spark.sql.functions.translate df.select(translate(col(
"Description"), "LEET", "1337"), col("Description")) .show(2)

# in Python from pyspark.sql.functions import translate df.select(translate(col(
"Description"), "LEET", "1337"), col("Description")) \ .show(2)

--in SQL
SELECT translate(Description, 'LEET', '1337'), Description FROM dfTable

+----------------------------------+--------------------+
|translate(Description, LEET, 1337)|         Description|
+----------------------------------+--------------------+
|           WHI73 HANGING H3A...|WHITE HANGING HEA...|
|            WHI73 M37A1 1AN73RN|WHITE METAL LANTERN|
+----------------------------------+--------------------+
```

We can also perform something similar, like pulling out the first mentioned color:

```
// in Scala import org.apache.spark.sql.functions.regexp_extract val regexString = si
mpleColors.map(_.toUpperCase).mkString("(", "|", ")")
// the | signifies OR in regular expression syntax df.select(    regexp_extract(col("Descri
ption"), regexString, 1).alias("color_clean"),    col("Description")).show(2)

# in Python from pyspark.sql.functions import regexp_extract extract_str = "(BLACK|WHI
TE|RED|GREEN|BLUE)" df.select(    regexp_extract(col("Description"), extract_str, 1).
alias("color_clean"),    col("Description")).show(2)

--in SQL
SELECT regexp_extract(Description, '(BLACK|WHITE|RED|GREEN|BLUE)', 1),
  Description
FROM dfTable

+-------------+--------------------+
| color_clean|         Description|
+-------------+--------------------+
|        WHITE|WHITE HANGING HEA...|
|        WHITE| WHITE METAL LANTERN|
+-------------+--------------------+
```

Sometimes, rather than extracting values, we simply want to check for their existence. We can do this with the contains method on each column. This will return a Boolean declaring whether the value you specify is in the column's string:

```
// in Scala val containsBlack = col("Description").contains("BLACK") val
containsWhite = col("DESCRIPTION").contains("WHITE") df.withCol
umn("hasSimpleColor", containsBlack.or(containsWhite))
  .where("hasSimpleColor")
  .select("Description").show(3, false)
```

In Python and SQL, we can use the i nst r function:

```
# i n Pyt hon f rom pyspark. sql . f unct i ons i mport i nst r cont ai nsBl ack = i nst
r( col ( "Descri pt i on") , "BLACK") >= 1 cont ai nsWhi t e = i nst r( col ( "Descri pt
i on") , "WHI TE") >= 1 df . wi t hCol umn( "hasSi mpl eCol or", cont ai nsBl ack |
cont ai nsWhi t e) \
  . where( "hasSi mpl eCol or") \
  . sel ect ( "Descri pt i on") . show( 3, Fal se)

- - i n SQL
SELECT Descri pt i on FROM df Tabl e
WHERE i nst r( Descri pt i on, ' BLACK' ) >= 1 OR i nst r( Descri pt i on, ' WHI TE' ) >= 1

+- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - + |Descri
pt i on                  |
+- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - +
|WHI TE HANGI NG HEART T- LI GHT HOLDER|
|WHI TE METAL LANTERN              |
|RED WOOLLY HOTTI E WHI TE HEART.    |
+- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - +
```

This is trivial with just two values, but it becomes more complicated when there are values.

Let's work through this in a more rigorous way and take advantage of Spark's ability to accept a dynamic number of arguments. When we convert a list of values into a set of arguments and pass them into a function, we use a language feature called varargs. Using this feature, we can effectively unravel an array of arbitrary length and pass it as arguments to a function. This, coupled with sel ect makes it possible for us to create arbitrary numbers of columns dynamically:

```
// i n Scal a val si mpl eCol ors = Seq( "bl ack", "whi t e", "red", "green", "bl ue")
val sel ect edCol umns = si mpl eCol ors. map( col or => {
  col ( "Descri pt i on") . cont ai ns( col or. t oUpperCase) . al i as( s"i s_$col or")
} ) : +expr( "*") // coul d al so append t hi s val ue df . sel ect ( sel ect edCol umns: _*) .
where( col ( "i s_whi t e") . or( col ( "i s_red") ) )   . sel ect ( "Descri pt i on") . show( 3, f al
se)

+- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - + |Descri
pt i on                  |
+- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - +
|WHI TE HANGI NG HEART T- LI GHT HOLDER|
|WHI TE METAL LANTERN              |
|RED WOOLLY HOTTI E WHI TE HEART.    |
+- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - +
```

We can also do this quite easily in Python. In this case, we're going to use a different function, l ocat e, that returns the integer location (1 based location). We then convert that to a Boolean before using it as the same basic feature:

```
# i n Pyt hon f rom pyspark. sql . f unct i ons i mport expr, l ocat e si mpl
eCol ors = [ "bl ack", "whi t e", "red", "green", "bl ue"] def col or_l ocat
or( col umn, col or_st ri ng) :
  ret urn l ocat e( col or_st ri ng. upper( ) , col umn) \
        . cast ( "bool ean") \          . al i as( "i s_" + c) sel ect edCol umns = [ col or_l ocat or( df .
Descri pt i on, c) f or c i n si mpl eCol ors] sel ect edCol umns. append( expr( "*") ) # has t o a
be Col umn t ype

df . sel ect ( *sel ect edCol umns) . where( expr( "i s_whi t e OR i s_red") ) \
  . sel ect ( "Descri pt i on") . show( 3, Fal se)
```

This simple feature can often help you programmatically generate columns or Boolean filters in a way that is simple to understand and extend. We could extend this to calculating the smallest common denominator for a given input value, or whether a number is a prime.

Working with Dates and Timestamps

Dates and times are a constant challenge in programming languages and databases. It's always necessary to keep track of timezones and ensure that formats are correct and valid. Spark does its best to keep things simple by focusing explicitly on two kinds of time-related information. There are dates, which focus exclusively on calendar dates, and timestamps, which include both date and time information. Spark, as we saw with our current dataset, will make a best effort to correctly identify column types, including dates and timestamps when we enable i nf erSchema. We can see that this worked quite well with our current dataset because it was able to identify and read our date format without us having to provide some specification for it.

As we hinted earlier, working with dates and timestamps closely relates to working with strings because we often store our timestamps or dates as strings and convert them into date types at runtime. This is less common when working with databases and structured data but much more common when we are working with text and CSV files. We will experiment with that shortly.

WARNING

There are a lot of caveats, unfortunately, when working with dates and timestamps, especially when it comes to timezone handling. In version 2.1 and before, Spark parsed according to the machine's timezone if timezones are not explicitly specified in the value that you are parsing. You can set a session local timezone if necessary by setting spark. conf . sessi onLocal Ti meZone in the SQL configurations. This should be set according to the Java TimeZone format. df . pri nt Schema( )

```
root
 |- - I nvoi ceNo: st ri ng ( nul l abl e = t rue)
 |- - St ockCode: st ri ng ( nul l abl e = t rue)
 |- - Descri pt i on: st ri ng ( nul l abl e = t rue)
 |- - Quant i t y: i nt eger ( nul l abl e = t rue)
 |- - I nvoi ceDat e: t i mest amp ( nul l abl e = t rue)
 |- - Uni t Pri ce: doubl e ( nul l abl e = t rue)
 |- - Cust omerI D: doubl e ( nul l abl e = t rue)
 |- - Count ry: st ri ng ( nul l abl e = t rue)
```

Although Spark will do read dates or times on a best-effort basis. However, sometimes there will be no getting around working with strangely formatted dates and times. The key to understanding the transformations that you are going to need to apply is to ensure that you know exactly what type and format you have at each given step of the way. Another common "gotcha" is that Spark's Ti mest ampType class supports only second-level precision, which means that if you're going to be working with milliseconds or microseconds, you'll need to work around this problem by potentially operating on them as l ongs. Any more precision when coercing to a Ti mest ampType will be removed.

Spark can be a bit particular about what format you have at any given point in time. It's important to be explicit when parsing or converting to ensure that there are no issues in doing so. At the end of the day, Spark is working with Java dates and timestamps and therefore conforms to those standards. Let's begin with the basics and get the current date and the current timestamps:

```
// i n Scal a
i mport org. apache. spark. sql . f unct i ons. { current _dat e, current _t i mest amp} val dat
eDF = spark. range( 10)
 . wi t hCol umn( "t oday", current _dat e( ) )  . wi t
hCol umn( "now", current _t i mest amp( ) ) dat eDF.
creat eOrRepl aceTempVi ew( "dat eTabl e")

# i n Pyt hon f rom pyspark. sql . f unct i ons i mport current _dat e, current _t i
mest amp dat eDF = spark. range( 10) \
 . wi t hCol umn( "t oday", current _dat e( ) ) \  . wi t
hCol umn( "now", current _t i mest amp( ) ) dat eDF.
creat eOrRepl aceTempVi ew( "dat eTabl e")
```

```
dat eDF. pri nt Schema( )

root
 |- - i d: l ong ( nul l abl e = f al se)
 |- - t oday: dat e ( nul l abl e = f al se)
 |- - now: t i mest amp ( nul l abl e = f al se)
```

Now that we have a simple DataFrame to work with, let's add and subtract five days from today. These functions take a column and then the number of days to either add or subtract as the arguments:

```
// i n Scal a i mport org. apache. spark. sql . f unct i ons. { dat e_add, dat e_sub} dat eDF. sel ect
( dat e_sub( col ( "t oday") , 5) , dat e_add( col ( "t oday") , 5) ) . show( 1)

# i n Pyt hon f rom pyspark. sql . f unct i ons i mport dat e_add, dat e_sub dat eDF. sel ect ( dat
e_sub( col ( "t oday") , 5) , dat e_add( col ( "t oday") , 5) ) . show( 1)

- - i n SQL
SELECT dat e_sub( t oday, 5) , dat e_add( t oday, 5) FROM dat eTabl e

+- - - - - - - - - - - - - - - - - +- - - - - - - - - - - - - - - - - +
|dat e_sub( t oday, 5) |dat e_add( t oday, 5) |
+- - - - - - - - - - - - - - - - - +- - - - - - - - - - - - - - - - - +
|      2017- 06- 12|      2017- 06- 22|
+- - - - - - - - - - - - - - - - - +- - - - - - - - - - - - - - - - - +
```

Another common task is to take a look at the difference between two dates. We can do this with the dat edi f f function that will return the number of days in between two dates. Most often we just care about the days, and because the number of days varies from month to month, there also exists a function, mont hs_bet ween, that gives you the number of months between two dates:

```
// i n Scal a i mport org. apache. spark. sql . f unct i ons. { dat edi f f , mont hs_bet ween, t
o_dat e} dat eDF. wi t hCol umn( "week_ago", dat e_sub( col ( "t oday") , 7) )
  . sel ect ( dat edi f f ( col ( "week_ago") , col ( "t oday") ) ) . show( 1) dat
eDF. sel ect (     t o_dat e( l i t ( "2016- 01- 01") ) . al i as( "st art ") ,     t
o_dat e( l i t ( "2017- 05- 22") ) . al i as( "end") )
  . sel ect ( mont hs_bet ween( col ( "st art ") , col ( "end") ) ) . show( 1)

# i n Pyt hon f rom pyspark. sql . f unct i ons i mport dat edi f f , mont hs_bet ween, t
o_dat e dat eDF. wi t hCol umn( "week_ago", dat e_sub( col ( "t oday") , 7) ) \   . sel
ect ( dat edi f f ( col ( "week_ago") , col ( "t oday") ) ) . show( 1)

dat eDF. sel ect (     t o_dat e( l i t ( "2016- 01- 01") ) . al i
as( "st art ") ,     t o_dat e( l i t ( "2017- 05- 22") ) . al i as(
"end") ) \
  . sel ect ( mont hs_bet ween( col ( "st art ") , col ( "end") ) ) . show( 1)
```

```
--in SQL
SELECT to_date('2016-01-01'), monthshs_between('2016-01-01','2017-01-01'), dat
edi ff('2016-01-01','2017-01-01') FROM dateTable

+--------------------------+|dat
edi ff(week_ago, today)|
+--------------------------+
|                -7|
+--------------------------+

+--------------------------+|mont
hs_between(start, end)|
+--------------------------+
|          -16.67741935|
+--------------------------+
```

Notice that we introduced a new function: the to_date function. The to_date function allows you to convert a string to a date, optionally with a specified format. We specify our format in the Java SimpleDateFormat which will be important to reference if you use this function:

```
// in Scala import org.apache.spark.sql.functions.{to_dat
e, lit} spark.range(5).withColumn("date", lit("2017-01-
01")) .select(to_date(col("date"))).show(1)

# in Python
from pyspark.sql.functions import to_date, lit spark.range(
5).withColumn("date", lit("2017-01-01")) \
 .select(to_date(col("date"))).show(1)
```

Spark will not throw an error if it cannot parse the date; rather, it will just return null. This can be a bit tricky in larger pipelines because you might be expecting your data in one format and getting it in another. To illustrate, let's take a look at the date format that has switched from year-

month-day to year-day-month. Spark will fail to parse this date and silently return null instead:

```
dateDF.select(to_date(lit("2016-20-12")),to_date(lit("2017-12-11"))).show(1)

+-------------------+-------------------+
|to_date(2016-20-12)|to_date(2017-12-11)|
+-------------------+-------------------
+|               null|       2017-12-11|
+-------------------+-------------------+
```

We find this to be an especially tricky situation for bugs because some dates might match the correct format, whereas others do not. In the previous example, notice how the second date appears as Decembers 11th instead of the correct day, November 12th. Spark doesn't throw an error because it cannot know whether the days are mixed up or that specific row is incorrect.

Let's fix this pipeline, step by step, and come up with a robust way to avoid these issues entirely. The first step is to remember that we need to specify our date format according to the Java Si mpl eDat eFormat standard.

We will use two functions to fix this: t o_dat e and t o_t i mest amp. The former optionally expects a format, whereas the latter requires one:

```
// i n Scal a i mport org. apache. spark. sql . f unct i ons. t o_dat e val dat
eFormat = "yyyy- dd- MM" val cl eanDat eDF = spark. range( 1) . sel ect (
t o_dat e( l i t ( "2017- 12- 11") , dat eFormat ) . al i as( "dat e") ,    t o_dat
e( l i t ( "2017- 20- 12") , dat eFormat ) . al i as( "dat e2") ) cl eanDat eDF.
creat eOrRepl aceTempVi ew( "dat eTabl e2")

# i n Pyt hon f rom pyspark. sql . f unct i ons i mport
t o_dat e dat eFormat = "yyyy- dd- MM" cl eanDat
eDF = spark. range( 1) . sel ect (
  t o_dat e( l i t ( "2017- 12- 11") , dat eFormat ) . al i as( "dat e") ,    t
o_dat e( l i t ( "2017- 20- 12") , dat eFormat ) . al i as( "dat e2") ) cl eanDat
eDF. creat eOrRepl aceTempVi ew( "dat eTabl e2")

- - i n SQL
SELECT t o_dat e( dat e, ' yyyy- dd- MM') , t o_dat e( dat e2, ' yyyy- dd- MM') , t o_dat e( dat e) FROM
dat eTabl e2

+- - - - - - - - - - +- - - - - - - - - - +
|    dat e|    dat e2|
+- - - - - - - - - - +- - - - - - - - - - + |2017-
11- 12|2017- 12- 20|
+- - - - - - - - - - +- - - - - - - - - - +
```

Now let's use an example of t o_t i mest amp, which always requires a format to be specified:

```
// i n Scal a i mport org. apache. spark. sql . f unct i ons. t o_t i mest amp cl eanDat
eDF. sel ect ( t o_t i mest amp( col ( "dat e") , dat eFormat ) ) . show( )

# i n Pyt hon f rom pyspark. sql . f unct i ons i mport t o_t i mest amp cl eanDat
eDF. sel ect ( t o_t i mest amp( col ( "dat e") , dat eFormat ) ) . show( )

- - i n SQL
SELECT t o_t i mest amp( dat e, ' yyyy- dd- MM') , t o_t i mest amp( dat e2, ' yyyy- dd- MM') FROM
dat eTabl e2
```

```
+---------------------------------+
|to_timestamp(`date`, 'yyyy-dd-MM')|
+---------------------------------+
|              2017-11-12 00:00:00|
+---------------------------------+
```

Casting between dates and timestamps is simple in all languages—in SQL, we would do it in the following way:

```
--in SQL
SELECT cast(to_date("2017-01-01", "yyyy-dd-MM") as timestamp)
```

After we have our date or timestamp in the correct format and type, comparing between them is actually quite easy. We just need to be sure to either use a date/timestamp type or specify our string according to the right format of yyyy-MM-dd if we're comparing a date: cleanDateDF.

```
filter(col("date2") > lit("2017-12-12")).show()
```

One minor point is that we can also set this as a string, which Spark parses to a literal:

```
cleanDateDF.filter(col("date2") > "'2017-12-12'").show()
```

> **WARNING**
>
> Implicit type casting is an easy way to shoot yourself in the foot, especially when dealing with null values or dates in different timezones or formats. We recommend that you parse them explicitly instead of relying on implicit conversions.

Working with Nulls in Data

As a best practice, you should always use nulls to represent missing or empty data in your DataFrames. Spark can optimize working with null values more than it can if you use empty strings or other values. The primary way of interacting with null values, at DataFrame scale, is to use the .na subpackage on a DataFrame. There are also several functions for performing operations and explicitly specifying how Spark should handle null values. For more information, see Chapter 5 (where we discuss ordering), and also refer back to "Working with Booleans".

> **WARNING**
>
> Nulls are a challenging part of all programming, and Spark is no exception. In our opinion, being explicit is always better than being implicit when handling null values. For instance, in this part of the book, we saw how we can define columns as having null types. However, this comes with a catch. When we declare a column as not having a null time, that is not actually *enforced*. To reiterate, when you define a schema in which all

columns are declared to *not* have null values, Spark will not enforce that and will happily let null values into that column. The nullable signal is simply to help Spark SQL optimize for handling that column. If you have null values in columns that should not have null values, you can get an incorrect result or see strange exceptions that can be difficult to debug.

There are two things you can do with null values: you can explicitly drop nulls or you can fill them with a value (globally or on a per-column basis). Let's experiment with each of these now.

Coalesce

Spark includes a function to allow you to select the first non-null value from a set of columns by using the coal esce function. In this case, there are no null values, so it simply returns the first column:

```
// i n Scal a i mport org. apache. spark. sql . f unct i ons. coal esce df . sel ect ( coal
esce( col ( "Descri pt i on") , col ( "Cust omerI d") ) ) . show( )

# i n Pyt hon f rom pyspark. sql . f unct i ons i mport
coal esce
df . sel ect ( coal esce( col ( "Descri pt i on") , col ( "Cust omerI d") ) ) . show( )
```

ifnull, nullIf, nvl, and nvl2

There are several other SQL functions that you can use to achieve similar things. i f nul l allows you to select the second value if the first is null, and defaults to the first. Alternatively, you could use nul l i f , which returns null if the two values are equal or else returns the second if they are not. nvl returns the second value if the first is null, but defaults to the first. Finally, nvl 2 returns the second value if the first is not null; otherwise, it will return the last specified value (el se_val ue in the following example):

```
- - i n SQL SELECT   i f nul l ( nul l , ' ret
urn_val ue' ) ,   nul l i f ( ' val ue' , ' val
ue' ) ,
 nvl ( nul l , ' ret urn_val ue' ) ,
 nvl 2( ' not _nul l ' , ' ret urn_val ue' , "el se_val ue") FROM df Tabl
e LI MI T 1

+- - - - - - - - - - - - +- - - - +- - - - - - - - - - - +- - - - - - - - - - - -
+ |       a|  b|       c|       d|
+- - - - - - - - - - - - +- - - - +- - - - - - - - - - - +- - - - - - - - - - - -+
|ret urn_val ue|nul l |ret urn_val ue|ret urn_val ue|
+- - - - - - - - - - - - +- - - - +- - - - - - - - - - - +- - - - - - - - - - - -+
```

Naturally, we can use these in select expressions on DataFrames, as well.

drop

The simplest function is drop, which removes rows that contain nulls. The default is to drop any row in which any value is null:

```
df . na. drop( ) df . na.
drop( "any")
```

In SQL, we have to do this column by column:

```
- - i n SQL
SELECT * FROM df Tabl e WHERE Descri pt i on I S NOT NULL
```

Specifying "any" as an argument drops a row if any of the values are null. Using “all” drops the row only if all values are nul l or NaN for that row: df . na. drop("al l ")

We can also apply this to certain sets of columns by passing in an array of columns:

```
// i n Scal a df . na. drop( "al l ", Seq( "St ockCode", "I nvoi
ceNo") )

# i n Pyt hon
df . na. drop( "al l ", subset =[ "St ockCode", "I nvoi ceNo"] ) fill
```

Using the f i l l function, you can fill one or more columns with a set of values. This can be done by specifying a map—that is a particular value and a set of columns.

For example, to fill all nul l values in columns of type String, you might specify the following: df

```
. na. f i l l ( "Al l Nul l val ues become t hi s st ri ng")
```

We could do the same for columns of type Integer by using df . na. f i l l (5: I nt eger) , or for Doubles df . na. f i l l (5: Doubl e) . To specify columns, we just pass in an array of column names like we did in the previous example:

```
// i n Scal a df . na. f i l l ( 5, Seq( "St ockCode", "I nvoi
ceNo") )

# i n Pyt hon
df . na. f i l l ( "al l ", subset =[ "St ockCode", "I nvoi ceNo"] )
```

We can also do this with with a Scala Map, where the key is the column name and the value is the value we would like to use to fill null values:

```
// in Scala val fillColValues = Map("StockCode" -> 5, "Description" -> "No Value")
df.na.fill(fillColValues)

# in Python fill_cols_vals = {"StockCode": 5, "Description" : "No Val
ue"} df.na.fill(fill_cols_vals)
```

replace

In addition to replacing null values like we did with drop and fill, there are more flexible options that you can use with more than just null values. Probably the most common use case is to replace all values in a certain column according to their current value. The only requirement is that this value be the same type as the original value:

```
// in Scala df.na.replace("Description", Map("" ->
"UNKNOWN"))

# in Python
df.na.replace([""], ["UNKNOWN"], "Description")
```

Ordering

As we discussed in Chapter 5, you can use asc_nulls_first, desc_nulls_first, asc_nulls_last, or desc_nulls_last to specify where you would like your null values to appear in an ordered DataFrame.

Working with Complex Types

Complex types can help you organize and structure your data in ways that make more sense for the problem that you are hoping to solve. There are three kinds of complex types: structs, arrays, and maps.

Structs

You can think of structs as DataFrames within DataFrames. A worked example will illustrate this more clearly. We can create a struct by wrapping a set of columns in parenthesis in a query: df.

```
selectExpr("(Description, InvoiceNo) as complex", "*")

df.selectExpr("struct(Description, InvoiceNo) as complex", "*")

// in Scala import org.apache.spark.sql.functions.struct val complexDF = df.select(struct
("Description", "InvoiceNo").alias("complex")) complexDF.createOrReplaceTempView(
"complexDF")
```

```
# i n Pyt hon f rom pyspark. sql . f unct i ons i mport st ruct compl exDF = df . sel ect ( st ruct (
"Descri pt i on", "I nvoi ceNo") . al i as( "compl ex") ) compl exDF. creat eOrRepl aceTempVi
ew( "compl exDF")
```

We now have a DataFrame with a column compl ex. We can query it just as we might another DataFrame, the only difference is that we use a dot syntax to do so, or the column method get Fi el d:

```
compl exDF. sel ect ( "compl ex. Descri pt i on")
compl exDF. sel ect ( col ( "compl ex") . get Fi el d( "Descri pt i on") )
```

We can also query all values in the struct by using *. This brings up all the columns to the toplevel DataFrame: compl exDF. sel ect ("compl ex. *")

```
- - i n SQL
SELECT compl ex. * FROM compl exDF
```

Arrays

To define arrays, let's work through a use case. With our current data, our objective is to take every single word in our Descri pt i on column and convert that into a row in our DataFrame.

The first task is to turn our Descri pt i on column into a complex type, an array.

split

We do this by using the spl i t function and specify the delimiter:

```
// i n Scal a i mport org. apache. spark. sql . f unct i ons.
spl i t df . sel ect ( spl i t ( col ( "Descri pt i on") , " ") ) .
show( 2)

# i n Pyt hon f rom pyspark. sql . f unct i ons i mport spl i t df .
sel ect ( spl i t ( col ( "Descri pt i on") , " ") ) . show( 2)

- - i n SQL
SELECT spl i t ( Descri pt i on, ' ' ) FROM df Tabl e

+- - - - - - - - - - - - - - - - - - - - + |spl
i t ( Descri pt i on,  ) |
+- - - - - - - - - - - - - - - - - - - - +
| [ WHI TE, HANGI NG, . . . |
| [ WHI TE, METAL, LA. . . |
+- - - - - - - - - - - - - - - - - - - - +
```

This is quite powerful because Spark allows us to manipulate this complex type as another column. We can also query the values of the array using Python-like syntax:

```
// in Scala df.select(split(col("Description"), " ").alias("array_col
")).selectExpr("array_col[0]").show(2)

# in Python df.select(split(col("Description"), " ").alias("array_col
")).\ .selectExpr("array_col[0]").show(2)

--in SQL
SELECT split(Description, ' ')[0] FROM dfTable
```

This gives us the following result:

```
+------------+
|array_col[0]|
+------------+
|       WHITE|
|       WHITE|+----
--------+
```

Array Length

We can determine the array's length by querying for its size:

```
// in Scala import org.apache.spark.sql.functions.size df.select(size(split(col(
"Description"), " "))).show(2) // shows 5 and 3

# in Python from pyspark.sql.functions i
mport size df.select(size(split(col("Descri
ption"), " "))).show(2) # shows 5 and 3
```

array_contains

We can also see whether this array contains a value:

```
// in Scala
import org.apache.spark.sql.functions.array_contains df.select(array_contains(spl
it(col("Description"), " "), "WHITE")).show(2)

# in Python from pyspark.sql.functions import array_contains df.select(array_cont
ains(split(col("Description"), " "), "WHITE")).show(2)

--in SQL
SELECT array_contains(split(Description, ' '), 'WHITE') FROM dfTable
```

This gives us the following result:

```
+----------------------------------------+ |array_cont
ai ns( spl i t ( Descri pt i on,  ) , WHI TE) |
+----------------------------------------+
|                        t rue|
|                        t rue|
+----------------------------------------+
```

However, this does not solve our current problem. To convert a complex type into a set of rows (one per value in our array), we need to use the expl ode function.

explode

The expl ode function takes a column that consists of arrays and creates one row (with the rest of the values duplicated) per value in the array. Figure 6-1 illustrates the process.

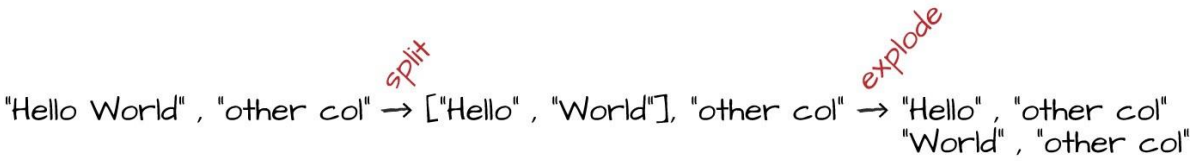


Figure 6-1. Exploding a column of text

```
// i n Scal a i mport org. apache. spark. sql . f unct i ons. { spl i t , expl
ode}

df . wi t hCol umn( "spl i t t ed", spl i t ( col ( "Descri pt i on") , " ") )  . wi
t hCol umn( "expl oded", expl ode( col ( "spl i t t ed") ) )
 . sel ect ( "Descri pt i on", "I nvoi ceNo", "expl oded") . show( 2)
# i n Pyt hon f rom pyspark. sql . f unct i ons i mport spl i t ,
expl ode

df . wi t hCol umn( "spl i t t ed", spl i t ( col ( "Descri pt i on") , " ") ) \  . wi
t hCol umn( "expl oded", expl ode( col ( "spl i t t ed") ) ) \
 . sel ect ( "Descri pt i on", "I nvoi ceNo", "expl oded") . show( 2)

- - i n SQL
SELECT Descri pt i on, I nvoi ceNo, expl oded
FROM ( SELECT *, spl i t ( Descri pt i on, " ") as spl i t t ed FROM df Tabl e) LATERAL
VI EW expl ode( spl i t t ed) as expl oded
```

This gives us the following result:

```
+--------------------+---------+--------+
|       Descri pt i on|I nvoi ceNo|expl oded|
+--------------------+---------+--------+
|WHI TE HANGI NG HEA. . . |   536365|   WHI TE|
|WHI TE HANGI NG HEA. . . |   536365| HANGI NG|
+--------------------+---------+--------+
```

Maps

Maps are created by using the map function and key-value pairs of columns. You then can select them just like you might select from an array:

```
// i n Scal a i mport org. apache. spark. sql . f unct i ons. map df . sel ect ( map( col ( "Descri pt i on") , col ( "I nvoi ceNo") ) . al i as( "compl ex_map") ) . show( 2)

# i n Pyt hon f rom pyspark. sql . f unct i ons i mport creat e_map df . sel ect ( creat e_map( col ( "Descri pt i on") , col ( "I nvoi ceNo") ) . al i as( "compl ex_map") ) \   . show( 2)

- - i n SQL
SELECT map( Descri pt i on, I nvoi ceNo) as compl ex_map FROM df Tabl e WHERE
Descri pt i on I S NOT NULL
```

This produces the following result:

```
+- - - - - - - - - - - - - - - - - - - +
|       compl ex_map|
+- - - - - - - - - - - - - - - - - - - +
|Map( WHI TE HANGI NG. . . |
|Map( WHI TE METAL L. . . |
+- - - - - - - - - - - - - - - - - - - +
```

You can query them by using the proper key. A missing key returns nul l :

```
// i n Scal a df . sel ect ( map( col ( "Descri pt i on") , col ( "I nvoi ceNo") ) . al i as( "compl ex_map") )   . sel ect Expr( "compl ex_map[ ' WHI TE METAL LANTERN' ] ") . show( 2)

# i n Pyt hon df . sel ect ( map( col ( "Descri pt i on") , col ( "I nvoi ceNo") ) . al i as( "compl ex_map") ) \
  . sel ect Expr( "compl ex_map[ ' WHI TE METAL LANTERN' ] ") . show( 2) This
```

gives us the following result:

```
+- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - + |compl
ex_map[ WHI TE METAL LANTERN] |
+- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - +
|                    nul l |
|                   536365|
+- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - +
```

You can also explode map types, which will turn them into columns:

```
// i n Scal a df . sel ect ( map( col ( "Descri pt i on") , col ( "I nvoi ceNo") ) . al i as( "compl ex_map") )   . sel ect Expr( "expl ode( compl ex_map) ") . show( 2)
```

```
# i n Pyt hon df . sel ect ( map( col ( "Descri pt i on") , col ( "I nvoi ceNo") ) . al i as( "compl
ex_map") ) \
  . sel ect Expr( "expl ode( compl ex_map) ") . show( 2)
```

This gives us the following result:

```
+- - - - - - - - - - - - - - - - - - - +- - - - - - +
|               key| val ue|
+- - - - - - - - - - - - - - - - - - - +- - - - - - +
|WHI TE HANGI NG HEA. . . |536365|
| WHI TE METAL LANTERN|536365| +-
- - - - - - - - - - - - - - - - - - - +- - - - - - +
```

Working with JSON

Spark has some unique support for working with JSON data. You can operate directly on strings of JSON in Spark and parse from JSON or extract JSON objects. Let's begin by creating a JSON column:

```
// i n Scal a val j sonDF = spark. range( 1) . sel ect Expr(
"""
 ' { "myJ SONKey" : { "myJ SONVal ue" : [ 1, 2, 3] } } ' as j sonSt ri ng""")

# i n Pyt hon j sonDF = spark. range( 1) . sel ect
Expr( """   ' { "myJ SONKey" : { "myJ SONVal ue"
: [ 1, 2, 3] } } ' as j sonSt ri ng""")
```

You can use the get _j son_obj ect to inline query a JSON object, be it a dictionary or array. You can use j son_t upl e if this object has only one level of nesting:

```
// i n Scal a
i       mport org. apache. spark. sql . f unct i ons. { get _j son_obj ect , j son_t upl e} j sonDF. sel ect (
get _j son_obj ect ( col ( "j sonSt ri ng") , "$. myJ SONKey. myJ SONVal ue[ 1] ") as "col umn",    j son_t
upl e( col ( "j sonSt ri ng") , "myJ SONKey") ) . show( 2)

# i n Pyt hon f rom pyspark. sql . f unct i ons i mport get _j son_obj ect , j son_t
upl e

j       sonDF. sel ect (    get _j son_obj ect ( col ( "j sonSt ri ng") , "$. myJ SONKey. myJ SONVal ue[ 1] ")
as "col umn",    j son_t upl e( col ( "j sonSt ri ng") , "myJ SONKey") ) . show( 2)
```

Here's the equivalent in SQL:

```
j sonDF. sel ect Expr(
  "j son_t upl e( j sonSt ri ng, ' $. myJ SONKey. myJ SONVal ue[ 1] ' ) as col umn") . show( 2)
```

This results in the following table:

```
+------+--------------------+ |col
umn|              c0|
+------+--------------------+
|    2|{ "myJ SONVal ue": [ 1. . . |
+------+--------------------+
```

You can also turn a StructType into a JSON string by using the t o_j son function:

```
// i n Scal a i mport org. apache. spark. sql . f unct i ons. t o_j son df
. sel ect Expr( "( I nvoi ceNo, Descri pt i on) as mySt ruct ")   . sel ect
( t o_j son( col ( "mySt ruct ") ) )

# i n Pyt hon f rom pyspark. sql . f unct i ons i mport
t o_j son
df . sel ect Expr( "( I nvoi ceNo, Descri pt i on) as mySt ruct ") \
  . sel ect ( t o_j son( col ( "mySt ruct ") ) )
```

This function also accepts a dictionary (map) of parameters that are the same as the JSON data source. You can use the f rom_j son function to parse this (or other JSON data) back in. This naturally requires you to specify a schema, and optionally you can specify a map of options, as well:

```
// i n Scal a i mport org. apache. spark. sql . f unct i ons. f rom_j son
i mport org. apache. spark. sql . t ypes. _ val parseSchema = new St
ruct Type( Array(    new St ruct Fi el d( "I nvoi ceNo", St ri ngType, t
rue) ,   new St ruct Fi el d( "Descri pt i on", St ri ngType, t rue) ) ) df
. sel ect Expr( "( I nvoi ceNo, Descri pt i on) as mySt ruct ")
  . sel ect ( t o_j son( col ( "mySt ruct ") ) . al i as( "newJ SON") )
  . sel ect ( f rom_j son( col ( "newJ SON") , parseSchema) , col ( "newJ SON") ) . show( 2)

# i n Pyt hon f rom pyspark. sql . f unct i ons i mport f
rom_j son f rom pyspark. sql . t ypes i mport *
parseSchema = St ruct Type( (
 St ruct Fi el d( "I nvoi ceNo", St ri ngType( ) , True) ,   St ruct Fi el d(
"Descri pt i on", St ri ngType( ) , True) ) ) df . sel ect Expr( "( I nvoi
ceNo, Descri pt i on) as mySt ruct ") \
  . sel ect ( t o_j son( col ( "mySt ruct ") ) . al i as( "newJ SON") ) \
  . sel ect ( f rom_j son( col ( "newJ SON") , parseSchema) , col ( "newJ SON") ) . show( 2) This
```

gives us the following result:

```
+----------------------+--------------------+
|j sont ost ruct s( newJ SON) |           newJ SON|
+----------------------+--------------------+
|  [ 536365, WHI TE HAN. . . |{ "I nvoi ceNo": "536. . . |
|  [ 536365, WHI TE MET. . . |{ "I nvoi ceNo": "536. . . |
+----------------------+--------------------+
```

User-Defined Functions

One of the most powerful things that you can do in Spark is define your own functions. These user-defined functions (UDFs) make it possible for you to write your own custom transformations using Python or Scala and even use external libraries. UDFs can take and return one or more columns as input. Spark UDFs are incredibly powerful because you can write them in several different programming languages; you do not need to create them in an esoteric format or domain-specific language. They're just functions that operate on the data, record by record. By default, these functions are registered as temporary functions to be used in that specific SparkSession or Context.

Although you can write UDFs in Scala, Python, or Java, there are performance considerations that you should be aware of. To illustrate this, we're going to walk through exactly what happens when you create UDF, pass that into Spark, and then execute code using that UDF.

The first step is the actual function. We'll create a simple one for this example. Let's write a power3 function that takes a number and raises it to a power of three:

```
// in Scala val udfExampleDF = spark.range(5).toDF(
"num") def power3(number:Double):Double =
number * number * number power3(2.0)

# in Python udfExampleDF = spark.range(5).t
oDF("num") def power3(doublevalue):  return
doublevalue ** 3 power3(2.0)
```

In this trivial example, we can see that our functions work as expected. We are able to provide an individual input and produce the expected result (with this simple test case). Thus far, our expectations for the input are high: it must be a specific type and cannot be a null value (see "Working with Nulls in Data").

Now that we've created these functions and tested them, we need to register them with Spark so that we can use them on all of our worker machines. Spark will serialize the function on the driver and transfer it over the network to all executor processes. This happens regardless of language.

When you use the function, there are essentially two different things that occur. If the function is written in Scala or Java, you can use it within the Java Virtual Machine (JVM). This means that there will be little performance penalty aside from the fact that you can't take advantage of code generation capabilities that Spark has for built-in functions. There can be performance issues if you create or use a lot of objects; we cover that in the section on optimization in Chapter 19.

If the function is written in Python, something quite different happens. Spark starts a Python process on the worker, serializes all of the data to a format that Python can understand (remember, it was in the JVM earlier), executes the function row by row on that data in the

Python process, and then finally returns the results of the row operations to the JVM and Spark.
Figure 6-2 provides an overview of the process.

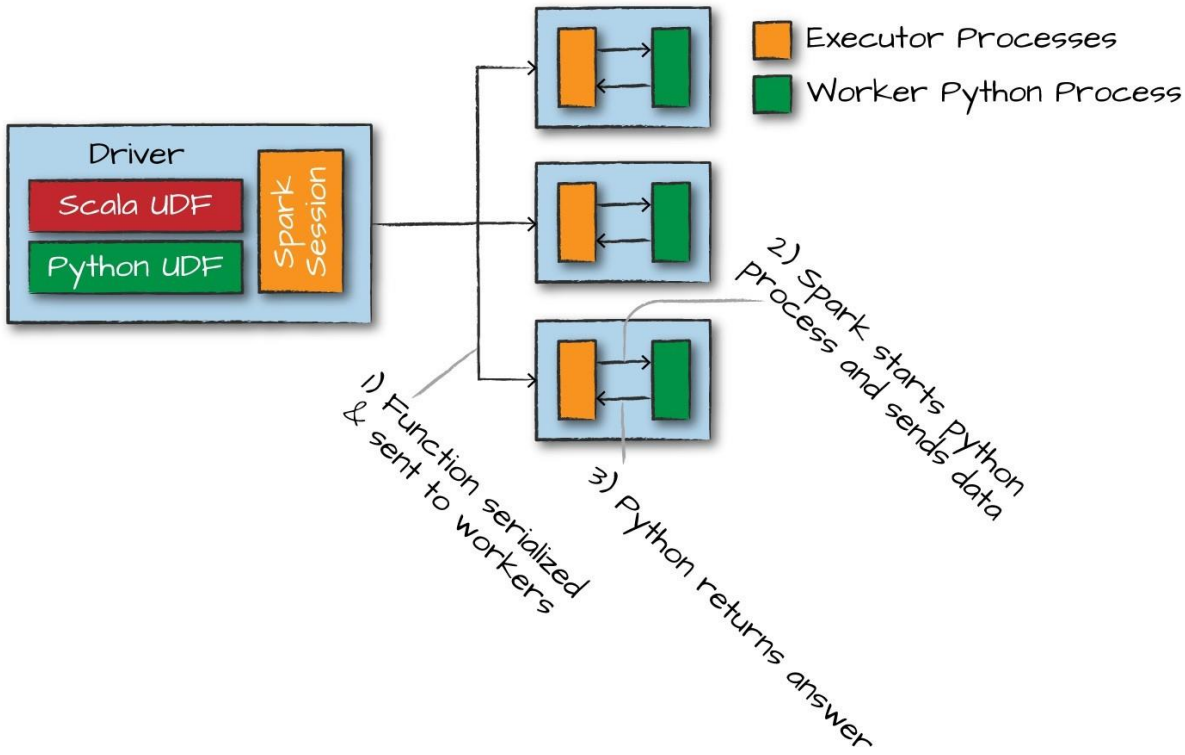


Figure 6-2. Figure caption

WARNING

Starting this Python process is expensive, but the real cost is in serializing the data to Python. This is costly for two reasons: it is an expensive computation, but also, after the data enters Python, Spark cannot manage the memory of the worker. This means that you could potentially cause a worker to fail if it becomes resource constrained (because both the JVM and Python are competing for memory on the same machine). We recommend that you write your UDFs in Scala or Java—the small amount of time it should take you to write the function in Scala will always yield significant speed ups, and on top of that, you can still use the function from Python!

Now that you have an understanding of the process, let's work through an example. First, we need to register the function to make it available as a DataFrame function:

```
// i n Scal a i mport org. apache. spark. sql . f unct i ons.
udf val power3udf = udf ( power3( _: Doubl e) : Doubl
e)
```

We can use that just like any other DataFrame function:

```
// in Scala udfExampleDF.select(power3udf(col("num")
)).show()
```

The same applies to Python—first, we register it:

```
# in Python from pyspark.sql.functions
import udf power3udf = udf(power3)
```

Then, we can use it in our DataFrame code:

```
# in Python from pyspark.sql.functions import col udf
ExampleDF.select(power3udf(col("num")))).show(2)

+-----------+
|power3(num)|
+-----------+
|          0|
|          1|
+-----------+
```

At this juncture, we can use this only as a DataFrame function. That is to say, we can't use it within a string expression, only on an expression. However, we can also register this UDF as a Spark SQL function. This is valuable because it makes it simple to use this function within SQL as well as across languages.

Let's register the function in Scala:

```
// in Scala spark.udf.register("power3", power3(_: Double) :
Double) udf ExampleDF.selectExpr("power3(num)").show(2)
```

Because this function is registered with Spark SQL—and we've learned that any Spark SQL function or expression is valid to use as an expression when working with DataFrames—we can turn around and use the UDF that we wrote in Scala, in Python. However, rather than using it as a DataFrame function, we use it as a SQL expression:

```
# in Python udfExampleDF.selectExpr("power3(num)
").show(2)
# registered in Scala
```

We can also register our Python function to be available as a SQL function and use that in any language, as well.

One thing we can also do to ensure that our functions are working correctly is specify a return type. As we saw in the beginning of this section, Spark manages its own type information, which does not align exactly with Python's types. Therefore, it's a best practice to define the return type for your function when you define it. It is important to note that specifying the return type is not necessary, but it is a best practice.

If you specify the type that doesn't align with the actual type returned by the function, Spark will not throw an error but will just return nul l to designate a failure. You can see this if you were to switch the return type in the following function to be a Doubl eType:

```
# i n Pyt hon f rom pyspark. sql . t ypes i mport I nt egerType, Doubl
eType spark. udf . regi st er( "power3py", power3, Doubl eType( ) )

# i n Pyt hon udf Exampl eDF. sel ect Expr( "power3py( num)
") . show( 2)
# regi st ered vi a Pyt hon
```

This is because the range creates integers. When integers are operated on in Python, Python won't convert them into floats (the corresponding type to Spark's double type), therefore we see null. We can remedy this by ensuring that our Python function returns a float instead of an integer and the function will behave correctly.

Naturally, we can use either of these from SQL, too, after we register them:

```
- - i n SQL
SELECT power3( 12) , power3py( 12) - - doesn' t work because of ret urn t ype
```

When you want to optionally return a value from a UDF, you should return None in Python and an Opt i on type in Scala:

```
## Hi ve UDFs
```

As a last note, you can also use UDF/UDAF creation via a Hive syntax. To allow for this, first you must enable Hive support when they create their SparkSession (via SparkSessi on. bui l der( ) . enabl eHi veSupport ( )). Then you can register UDFs in SQL. This is only supported with precompiled Scala and Java packages, so you'll need to specify them as a dependency:

```
- - i n SQL
CREATE TEMPORARY FUNCTI ON myFunc AS ' com. organi zat i on. hi ve. udf . Funct i onName'
```

Additionally, you can register this as a permanent function in the Hive Metastore by removing TEMPORARY.

Conclusion

This chapter demonstrated how easy it is to extend Spark SQL to your own purposes and do so in a way that is not some esoteric, domain-specific language but rather simple functions that are easy to test and maintain without even using Spark! This is an amazingly powerful tool that you can use to specify sophisticated business logic that can run on five rows on your local machines or on terabytes of data on a 100-node cluster!

Chapter 7. Aggregations

Aggregating is the act of collecting something together and is a cornerstone of big data analytics. In an aggregation, you will specify a *key* or *grouping* and an *aggregation function* that specifies how you should transform one or more columns. This function must produce one result for each group, given multiple input values. Spark's aggregation capabilities are sophisticated and mature, with a variety of different use cases and possibilities. In general, you use aggregations to summarize numerical data usually by means of some grouping. This might be a summation, a product, or simple counting. Also, with Spark you can aggregate any kind of value into an array, list, or map, as we will see in "Aggregating to Complex Types".

In addition to working with any type of values, Spark also allows us to create the following groupings types:

- The simplest grouping is to just summarize a complete DataFrame by performing an aggregation in a select statement.
- A "group by" allows you to specify one or more keys as well as one or more aggregation functions to transform the value columns.
- A "window" gives you the ability to specify one or more keys as well as one or more aggregation functions to transform the value columns. However, the rows input to the function are somehow related to the current row.
- A "grouping set," which you can use to aggregate at multiple different levels. Grouping sets are available as a primitive in SQL and via rollups and cubes in DataFrames.
- A "rollup" makes it possible for you to specify one or more keys as well as one or more aggregation functions to transform the value columns, which will be summarized hierarchically.
- A "cube" allows you to specify one or more keys as well as one or more aggregation functions to transform the value columns, which will be summarized across all combinations of columns.

Each grouping returns a Rel at i onal GroupedDat aset on which we specify our aggregations.

> **NOTE**
>
> An important thing to consider is how exact you need an answer to be. When performing calculations over big data, it can be quite expensive to get an *exact* answer to a question, and it's often much cheaper to simply request an approximate to a reasonable degree of accuracy. You'll note that we mention some approximation functions throughout the book and oftentimes this is a good opportunity to improve the speed and execution of your Spark jobs, especially for interactive and ad hoc analysis.

Let's begin by reading in our data on purchases, repartitioning the data to have far fewer partitions (because we know it's a small volume of data stored in a lot of small files), and caching the results for rapid access:

```
// in Scala val df = spark.read.format(
"csv")  .option("header", "true")
 .option("inferSchema", "true")
 .load("/data/retail-data/all/*.csv")
 .coalesce(5).df.cache().df.createOrRepl
aceTempView("dfTable")

# in Python df = spark.read.format
("csv").\  .option("header", "true")
\
 .option("inferSchema", "true").\
 .load("/data/retail-data/all/*.csv").\
 .coalesce(5).df.
cache()
df.createOrReplaceTempView("dfTable")
```

Here's a sample of the data so that you can reference the output of some of the functions:

```
+---------+---------+--------------------+--------+--------------+---------+---
--
|InvoiceNo|StockCode|       Description|Quantity|  InvoiceDate|UnitPrice|Cu...+--------
-+---------+--------------------+--------+--------------+---------+-----
|   536365|   85123A|WHITE HANGING...    |       6|12/1/2010 8:26|     2.55|  ...|   536365|
71053|WHITE METAL...     |       6|12/1/2010 8:26|    3.39| ......
|   536367|   21755|LOVE BUILDING BLO...|      3|12/1/2010 8:34|    5.95|  ...
|   536367|   21777|RECIPE BOX WITH M...|      4|12/1/2010 8:34|    7.95|  ...+---------+---
------+--------------------+--------+--------------+---------+-----
```

As mentioned, basic aggregations apply to an entire DataFrame. The simplest example is the count method: `df.count() == 541909`

If you've been reading this book chapter by chapter, you know that count is actually an action as opposed to a transformation, and so it returns immediately. You can use count to get an idea of the total size of your dataset but another common pattern is to use it to cache an entire DataFrame in memory, just like we did in this example.

Now, this method is a bit of an outlier because it exists as a method (in this case) as opposed to a function and is eagerly evaluated instead of a lazy transformation. In the next section, we will see count used as a lazy function, as well.

Aggregation Functions

All aggregations are available as functions, in addition to the special cases that can appear on DataFrames or via . st at , like we saw in Chapter 6. You can find most aggregation functions in the org. apache. spark. sql . f unct i ons package.

> **NOTE**
>
> There are some gaps between the available SQL functions and the functions that we can import in Scala and Python. This changes every release, so it's impossible to include a definitive list. This section covers the most common functions.

count

The first function worth going over is count , except in this example it will perform as a transformation instead of an action. In this case, we can do one of two things: specify a specific column to count, or all the columns by using count (*) or count (1) to represent that we want to count every row as the literal one, as shown in this example:

```
// i n Scal a i mport org. apache. spark. sql . f unct i ons.
count df . sel ect ( count ( "St ockCode") ) . show( ) //
541909

# i n Pyt hon f rom pyspark. sql . f unct i ons i mport count
df . sel ect ( count ( "St ockCode") ) . show( ) # 541909

- - i n SQL
SELECT COUNT( *) FROM df Tabl e
```

> **WARNING**
>
> There are a number of gotchas when it comes to null values and counting. For instance, when performing a count (*) , Spark will count null values (including rows containing all nulls). However, when counting an individual column, Spark will not count the null values.

countDistinct

Sometimes, the total number is not relevant; rather, it's the number of unique groups that you want. To get this number, you can use the count Di st i nct function. This is a bit more relevant for individual columns:

```
// i n Scal a i mport org. apache. spark. sql . f unct i ons. count Di
st i nct df . sel ect ( count Di st i nct ( "St ockCode") ) . show( ) //
4070
```

```
# in Python from pyspark.sql.functions import countDistinct
countDistinct df.select("StockCode")).show() # 4070

--in SQL
SELECT COUNT(DISTINCT *) FROM DFTABLE
```

approx_count_distinct

Often, we find ourselves working with large datasets and the exact distinct count is irrelevant. There are times when an approximation to a certain degree of accuracy will work just fine, and for that, you can use the approx_count_distinct function:

```
// in Scala import org.apache.spark.sql.functions.approx_count_distinct
df.select(approx_count_distinct("StockCode", 0.1)).show() // 3364

# in Python from pyspark.sql.functions import approx_count_distinct df.
select(approx_count_distinct("StockCode", 0.1)).show() # 3364

--in SQL
SELECT approx_count_distinct(StockCode, 0.1) FROM DFTABLE
```

You will notice that approx_count_distinct took another parameter with which you can specify the maximum estimation error allowed. In this case, we specified a rather large error and thus receive an answer that is quite far off but does complete more quickly than countDistinct. You will see much greater performance gains with larger datasets.

first and last

You can get the first and last values from a DataFrame by using these two obviously named functions. This will be based on the rows in the DataFrame, not on the values in the DataFrame:

```
// in Scala import org.apache.spark.sql.functions.{first, last}
df.select(first("StockCode"), last("StockCode")).show()

# in Python from pyspark.sql.functions import first, last df.sel
ect(first("StockCode"), last("StockCode")).show()

--in SQL
SELECT first(StockCode), last(StockCode) FROM dfTable
+-----------------------+----------------------+
|first(StockCode, false)|last(StockCode, false)|
+-----------------------+----------------------
+|           85123A|            22138|
+-----------------------+----------------------+
```

min and max

To extract the minimum and maximum values from a DataFrame, use the mi n and max functions:

```
// i n Scal a i mport org. apache. spark. sql . f unct i ons. { mi n,
max} df . sel ect ( mi n( "Quant i t y") , max( "Quant i t y") ) .
show( )

# i n Pyt hon f rom pyspark. sql . f unct i ons i mport mi n, max
df . sel ect ( mi n( "Quant i t y") , max( "Quant i t y") ) . show( )

- - i n SQL
SELECT mi n( Quant i t y) , max( Quant i t y) FROM df Tabl e
```

```
+- - - - - - - - - - - - - +- - - - - - - - - - - - - +
|mi n( Quant i t y) |max( Quant i t y) |
+- - - - - - - - - - - - - +- - - - - - - - - - - - - +
|     - 80995|      80995|
+- - - - - - - - - - - - - +- - - - - - - - - - - - - +
```

sum

Another simple task is to add all the values in a row using the sum function:

```
// i n Scal a i mport org. apache. spark. sql . f unct i ons.
sum  df . sel ect ( sum( "Quant i t y") ) . show( ) //
5176450

# i n Pyt hon f rom pyspark. sql . f unct i ons i mport
sum  df . sel ect ( sum( "Quant i t y") ) . show( ) #
5176450

- - i n SQL
SELECT sum( Quant i t y) FROM df Tabl e
```

sumDistinct

In addition to summing a total, you also can sum a distinct set of values by using the sumDi st i nct function:

```
// i n Scal a i mport org. apache. spark. sql . f unct i ons. sumDi
st i nct df . sel ect ( sumDi st i nct ( "Quant i t y") ) . show( ) //
29310

# i n Pyt hon f rom pyspark. sql . f unct i ons i mport sumDi st i
nct df . sel ect ( sumDi st i nct ( "Quant i t y") ) . show( ) # 29310

- - i n SQL
```

```
SELECT SUM( Quant i t y) FROM df Tabl e - - 29310
```

avg

Although you can calculate average by dividing sum by count , Spark provides an easier way to get that value via the avg or mean functions. In this example, we use al i as in order to more easily reuse these columns later:

```
// i n Scal a i mport org. apache. spark. sql . f unct i ons. { sum, count , avg,
expr}

df . sel ect (     count ( "Quant i t y") . al i as( "t ot al _t ransact i
ons") ,    sum( "Quant i t y") . al i as( "t ot al _purchases") ,    avg(
"Quant i t y") . al i as( "avg_purchases") ,    expr( "mean( Quant
i t y) ") . al i as( "mean_purchases") )
 . sel ect Expr(
  "t ot al _purchases/t ot al _t ransact i ons",
  "avg_purchases",
  "mean_purchases") . show( )

# i n Pyt hon f rom pyspark. sql . f unct i ons i mport sum, count , avg,
expr

df . sel ect (     count ( "Quant i t y") . al i as( "t ot al _t ransact i
ons") ,    sum( "Quant i t y") . al i as( "t ot al _purchases") ,    avg(
"Quant i t y") . al i as( "avg_purchases") ,    expr( "mean( Quant i t
y) ") . al i as( "mean_purchases") ) \
 . sel ect Expr(
  "t ot al _purchases/t ot al _t ransact i ons",
  "avg_purchases",
  "mean_purchases") . show( )

+- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - +- - - - - - - - - - - - - - - +- - - - - - - - - - - - - - -
+ |( t ot al _purchases / t ot al _t ransact i ons) |   avg_purchases|  mean_purchases|
+- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - +- - - - - - - - - - - - - - - +- - - - - - - - - - - - - - -
- + |                9. 55224954743324|9. 55224954743324|9. 55224954743324|
+- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - +- - - - - - - - - - - - - - - +- - - - - - - - - - - - - - - +
```

> **NOTE**
>
> You can also average all the distinct values by specifying distinct. In fact, most aggregate functions support doing so only on distinct values.

Variance and Standard Deviation

Calculating the mean naturally brings up questions about the variance and standard deviation. These are both measures of the spread of the data around the mean. The variance is the

average of the squared differences from the mean, and the standard deviation is the square root of the variance. You can calculate these in Spark by using their respective functions. However, something to note is that Spark has both the formula for the sample standard deviation as well as the formula for the population standard deviation. These are fundamentally different statistical formulae, and we need to differentiate between them. By default, Spark performs the formula for the sample standard deviation or variance if you use the vari ance or st ddev functions.

You can also specify these explicitly or refer to the population standard deviation or variance:

```
// i n Scal a i mport org. apache. spark. sql . f unct i ons. { var_pop, st
ddev_pop} i mport org. apache. spark. sql . f unct i ons. { var_samp, st
ddev_samp} df . sel ect ( var_pop( "Quant i t y") , var_samp( "Quant i t y") ,
st ddev_pop( "Quant i t y") , st ddev_samp( "Quant i t y") ) . show( )

# i n Pyt hon f rom pyspark. sql . f unct i ons i mport var_pop, st ddev_pop
f rom pyspark. sql . f unct i ons i mport var_samp, st ddev_samp df . sel
ect ( var_pop( "Quant i t y") , var_samp( "Quant i t y") ,   st ddev_pop(
"Quant i t y") , st ddev_samp( "Quant i t y") ) . show( )

- - i n SQL
SELECT var_pop( Quant i t y) , var_samp( Quant i t y) ,   st
ddev_pop( Quant i t y) , st ddev_samp( Quant i t y)
FROM df Tabl e
+- - - - - - - - - - - - - - - - - +- - - - - - - - - - - - - - - - - +- - - - - - - - - - - - - - - - - - +- - - - - - - - - - - - - - - - -
- + | var_pop( Quant i t y) |var_samp( Quant i t y) |st ddev_pop( Quant i t y) |st ddev_samp( Quan. . .
|
+- - - - - - - - - - - - - - - - - +- - - - - - - - - - - - - - - - - +- - - - - - - - - - - - - - - - - - +- - - - - - - - - - - - - - - - -
- + |47559. 303646609056|47559. 391409298754|  218. 08095663447796|   218. 081157850. . . |
+- - - - - - - - - - - - - - - - - +- - - - - - - - - - - - - - - - - +- - - - - - - - - - - - - - - - - - +- - - - - - - - - - - - - - - - -
- +
```

skewness and kurtosis

Skewness and kurtosis are both measurements of extreme points in your data. Skewness measures the asymmetry of the values in your data around the mean, whereas kurtosis is a measure of the tail of data. These are both relevant specifically when modeling your data as a probability distribution of a random variable. Although here we won't go into the math behind these specifically, you can look up definitions quite easily on the internet. You can calculate these by using the functions:

```
i mport org. apache. spark. sql . f unct i ons. { skewness, kurt osi s} df . sel
ect ( skewness( "Quant i t y") , kurt osi s( "Quant i t y") ) . show( )

# i n Pyt hon f rom pyspark. sql . f unct i ons i mport skewness, kurt osi s df .
sel ect ( skewness( "Quant i t y") , kurt osi s( "Quant i t y") ) . show( )
```

```
--in SQL
SELECT skewness(Quantity), kurtosis(Quantity) FROM dfTable

+------------------+------------------+
|skewness(Quantity)|kurtosis(Quantity)|
+------------------+------------------+
|-0.264075576105256|119768.05495536952|
+------------------+------------------+
```

Covariance and Correlation

We discussed single column aggregations, but some functions compare the interactions of the values in two difference columns together. Two of these functions are cov and corr, for covariance and correlation, respectively. Correlation measures the Pearson correlation coefficient, which is scaled between –1 and +1. The covariance is scaled according to the inputs in the data.

Like the var function, covariance can be calculated either as the sample covariance or the population covariance. Therefore it can be important to specify which formula you want to use. Correlation has no notion of this and therefore does not have calculations for population or sample. Here's how they work:

```
// in Scala import org.apache.spark.sql.functions.{corr, covar_pop,
covar_samp} df.select(corr("InvoiceNo", "Quantity"), covar_samp("Invoi
ceNo", "Quantity"), covar_pop("InvoiceNo", "Quantity")).show()

# in Python from pyspark.sql.functions import corr, covar_pop, covar_samp df.select(corr(
"InvoiceNo", "Quantity"), covar_samp("InvoiceNo", "Quantity"), covar_pop("Invoi
ceNo", "Quantity")).show()

--in SQL
SELECT corr(InvoiceNo, Quantity), covar_samp(InvoiceNo, Quantity),
covar_pop(InvoiceNo, Quantity) FROM dfTable

+-------------------------+-----------------------------+-------------------
--+|corr(InvoiceNo, Quantity)|covar_samp(InvoiceNo, Quantity)|covar_pop(InvoiceN...|
+-------------------------+-----------------------------+-------------------
--+|    4.912186085635685E-4|          1052.7280543902734|         1052.7...|
+-------------------------+-----------------------------+-------------------
--+
```

Aggregating to Complex Types

In Spark, you can perform aggregations not just of numerical values using formulas, you can also perform them on complex types. For example, we can collect a list of values present in a given column or only the unique values by collecting to a set.

You can use this to carry out some more programmatic access later on in the pipeline or pass the entire collection in a user-defined function (UDF):

```
// in Scala import org.apache.spark.sql.functions.{collect_set, collect_list}.agg(collect_set("Country"), collect_list("Country")).show()

# in Python from pyspark.sql.functions import collect_set, collect_list
df.agg(collect_set("Country"), collect_list("Country")).show()

--in SQL
SELECT collect_set(Country), collect_set(Country) FROM dfTable

+--------------------+---------------------+
|collect_set(Country)|collect_list(Country)|
+--------------------+---------------------+
|[Portugal, Italy,...|[United Kingdom, ...|
+--------------------+---------------------+
```

Grouping

Thus far, we have performed only DataFrame-level aggregations. A more common task is to perform calculations based on *groups* in the data. This is typically done on categorical data for which we group our data on one column and perform some calculations on the other columns that end up in that group.

The best way to explain this is to begin performing some groupings. The first will be a count, just as we did before. We will group by each unique invoice number and get the count of items on that invoice. Note that this returns another DataFrame and is lazily performed.

We do this grouping in two phases. First we specify the column(s) on which we would like to group, and then we specify the aggregation(s). The first step returns a RelationalGroupedDataset, and the second step returns a DataFrame.

As mentioned, we can specify any number of columns on which we want to group:

```
df.groupBy("InvoiceNo", "CustomerId").count().show()

--in SQL
SELECT count(*) FROM dfTable GROUP BY InvoiceNo, CustomerId

+---------+----------+-----+
|InvoiceNo|CustomerId|count|
+---------+----------+-----+
|   536846|     14573|   76|...
|  C544318|     12989|    1|
+---------+----------+-----+
```

Grouping with Expressions

As we saw earlier, counting is a bit of a special case because it exists as a method. For this, usually we prefer to use the count function. Rather than passing that function as an expression into a sel ect statement, we specify it as within agg. This makes it possible for you to pass-in arbitrary expressions that just need to have some aggregation specified. You can even do things like al i as a column after transforming it for later use in your data flow:

```
// i n Scal a i mport org. apache. spark. sql . f unct i ons.
count

df . groupBy( "I nvoi ceNo") . agg(   count (
"Quant i t y") . al i as( "quan") ,    expr(
"count ( Quant i t y) ") ) . show( )

# i n Pyt hon f rom pyspark. sql . f unct i ons i
mport count

df . groupBy( "I nvoi ceNo") . agg(     count (
"Quant i t y") . al i as( "quan") ,    expr( "count
( Quant i t y) ") ) . show( ) +---------+----
+---------------+
|I nvoi ceNo|quan|count ( Quant i t y) |
+---------+----+---------------+
|  536596|  6|         6| . . .
| C542604|  8|         8| +---------+-
---+---------------+
```

Grouping with Maps

Sometimes, it can be easier to specify your transformations as a series of Maps for which the key is the column, and the value is the aggregation function (as a string) that you would like to perform. You can reuse multiple column names if you specify them inline, as well:

```
// i n Scal a
df . groupBy( "I nvoi ceNo") . agg( "Quant i t y"- >"avg", "Quant i t y"- >"st ddev_pop") . show( )

# i n Pyt hon df . groupBy( "I nvoi ceNo") . agg( expr( "avg( Quant i t y) ") , expr( "st ddev_pop( Quant
i t y) ") ) \   . show( )

- - i n SQL
SELECT avg( Quant i t y) , st ddev_pop( Quant i t y) , I nvoi ceNo FROM df Tabl e
GROUP BY I nvoi ceNo

+---------+------------------+--------------------+
|I nvoi ceNo|     avg( Quant i t y) |st ddev_pop( Quant i t y) |
+---------+------------------+--------------------+
```

```
|  536596|          1.5|  1.1180339887498947| . . .
| C542604|          -8.0|  15.173990905493518|
+--------+-----------------+--------------------+
```

Window Functions

You can also use *window functions* to carry out some unique aggregations by either computing some aggregation on a specific "window" of data, which you define by using a reference to the current data. This window specification determines which rows will be passed in to this function. Now this is a bit abstract and probably similar to a standard group-by, so let's differentiate them a bit more.

A *group-by* takes data, and every row can go only into one grouping. A window function calculates a return value for every input row of a table based on a group of rows, called a frame. Each row can fall into one or more frames. A common use case is to take a look at a rolling average of some value for which each row represents one day. If you were to do this, each row would end up in seven different frames. We cover defining frames a little later, but for your reference, Spark supports three kinds of window functions: ranking functions, analytic functions, and aggregate functions.

Figure 7-1 illustrates how a given row can fall into multiple frames.

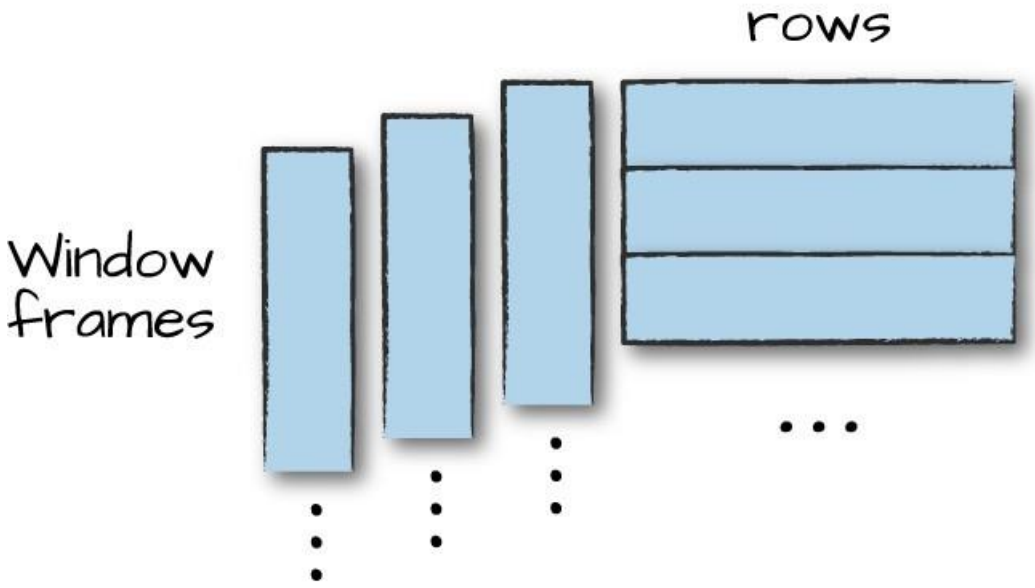


Figure 7-1. Visualizing window functions

To demonstrate, we will add a date column that will convert our invoice date into a column that contains only date information (not time information, too):

```
// in Scala import org.apache.spark.sql.functions.{col, t
o_date}
val dfWithDate = df.withColumn("date", to_date(col("InvoiceDate"),
```

```
  "MM/d/yyyy H: mm") ) df Wi t hDat e. creat eOrRepl
aceTempVi ew( "df Wi t hDat e")

# i n Pyt hon f rom pyspark. sql . f unct i ons i mport col , t o_dat e df Wi t hDat e = df . wi t hCol umn(
"dat e", t o_dat e( col ( "I nvoi ceDat e") , "MM/d/yyyy H: mm") ) df Wi t hDat e. creat eOrRepl
aceTempVi ew( "df Wi t hDat e")
```

The first step to a window function is to create a window specification. Note that the part i t i on by is unrelated to the partitioning scheme concept that we have covered thus far. It's just a similar concept that describes how we will be breaking up our group. The ordering determines the ordering within a given partition, and, finally, the frame specification (the rowsBet ween statement) states which rows will be included in the frame based on its reference to the current input row. In the following example, we look at all previous rows up to the current row:

```
// i n Scal a i mport org. apache. spark. sql . expressi ons.
Wi ndow i mport org. apache. spark. sql . f unct i ons. col
val wi ndowSpec = Wi ndow
 . part i t i onBy( "Cust omerI d", "dat e")
 . orderBy( col ( "Quant i t y") . desc)
 . rowsBet ween( Wi ndow. unboundedPrecedi ng, Wi ndow. current Row)

# i n Pyt hon f rom pyspark. sql . wi ndow i mport
Wi ndow f rom pyspark. sql . f unct i ons i mport
desc wi ndowSpec = Wi ndow\
 . part i t i onBy( "Cust omerI d", "dat e") \
 . orderBy( desc( "Quant i t y") ) \
 . rowsBet ween( Wi ndow. unboundedPrecedi ng, Wi ndow. current Row)
```

Now we want to use an aggregation function to learn more about each specific customer. An example might be establishing the maximum purchase quantity over all time. To answer this, we use the same aggregation functions that we saw earlier by passing a column name or expression. In addition, we indicate the window specification that defines to which frames of data this function will apply:

```
i mport org. apache. spark. sql . f unct i ons. max val maxPurchaseQuant i t y =
max( col ( "Quant i t y") ) . over( wi ndowSpec)

# i n Pyt hon f rom pyspark. sql . f unct i ons i
mport max
maxPurchaseQuant i t y = max( col ( "Quant i t y") ) . over( wi ndowSpec)
```

You will notice that this returns a column (or expressions). We can now use this in a DataFrame select statement. Before doing so, though, we will create the purchase quantity rank. To do that we use the dense_rank function to determine which date had the maximum

purchase quantity for every customer. We use dense_rank as opposed to rank to avoid gaps in the ranking sequence when there are tied values (or in our case, duplicate rows):

```
// i n Scal a i mport org. apache. spark. sql . f unct i ons. { dense_rank,
rank} val purchaseDenseRank = dense_rank( ) . over( wi ndowSpec) val
purchaseRank = rank( ) . over( wi ndowSpec)

# i n Pyt hon f rom pyspark. sql . f unct i ons i mport dense_rank,
rank purchaseDenseRank = dense_rank( ) . over( wi ndowSpec)
purchaseRank = rank( ) . over( wi ndowSpec)
```

This also returns a column that we can use in select statements. Now we can perform a sel ect to view the calculated window values:

```
// i n Scal a i mport org. apache. spark. sql . f unct i
ons. col

df Wi t hDat e. where( "Cust omerI d I S NOT NULL") . orderBy( "Cust omerI d")
 . sel ect ( col ( "Cust
omerI d") , col ( "dat e") ,
col ( "Quant i t y") ,
purchaseRank. al i as(
"quant i t yRank") ,
purchaseDenseRank. al i
as( "quant i t yDenseRank")
, maxPurchaseQuant i t y.
al i as( "maxPurchaseQuant
i t y") ) . show( )

# i n Pyt hon f rom pyspark. sql . f unct i ons i
mport col

df Wi t hDat e. where( "Cust omerI d I S NOT NULL") . orderBy( "Cust omerI d") \
 . sel ect ( col ( "Cust
omerI d") , col ( "dat e") ,
col ( "Quant i t y") ,
  purchaseRank. al i as( "quant i t yRank") , purchaseDenseRank. al i as(
"quant i t yDenseRank") , maxPurchaseQuant i t y. al i as(
"maxPurchaseQuant i t y") ) . show( )

- - i n SQL
SELECT Cust omerI d, dat e, Quant i t y,
 rank( Quant i t y) OVER ( PARTI TI ON BY Cust omerI d, dat e
                ORDER BY Quant i t y DESC NULLS LAST
                ROWS BETWEEN
                 UNBOUNDED PRECEDI NG AND CURRENT
ROW) as rank,

 dense_rank( Quant i t y) OVER ( PARTI TI ON BY Cust omerI d, dat e
```

```
                    ORDER BY Quant i t y DESC NULLS LAST
                    ROWS BETWEEN
                     UNBOUNDED PRECEDI NG AND                    CURRENT
ROW) as dRank,

 max( Quant i t y) OVER ( PARTI TI ON BY Cust omerI d, dat e
              ORDER BY Quant i t y DESC NULLS LAST
              ROWS BETWEEN
               UNBOUNDED PRECEDI NG AND
               CURRENT ROW) as maxPurchase
FROM df Wi t hDat e WHERE Cust omerI d I S NOT NULL ORDER BY Cust omerI d

+- - - - - - - - - -+- - - - - - - - - -+- - - - - - - -+- - - - - - - - - - - -+- - - - - - - - - - - - - - - -+- - - - - - - - - - - -
- + |Cust omerI d|     dat e|Quant i t y|quant i t yRank|quant i t yDenseRank|maxP. . . Quant i t y|
+- - - - - - - - - -+- - - - - - - - - -+- - - - - - - -+- - - - - - - - - - - -+- - - - - - - - - - - - - - - -+- - - - - - - - - - - -
- - +
|   12346|2011- 01- 18|  74215|       1|          1|      74215|
|   12346|2011- 01- 18|  - 74215|       2|          2|      74215|
|   12347|2010- 12- 07|    36|       1|          1|       36| |   12347|2010- 12- 07|    30|
2|          2|         36| . . .
|   12347|2010- 12- 07|    12|       4|          4|       36|
|   12347|2010- 12- 07|     6|      17|          5|       36|
|   12347|2010- 12- 07|     6|      17|          5|       36| +- - - - - - - - - -+- - - - - - - - - -+- - - - - - -
- +- - - - - - - - - - - -+- - - - - - - - - - - - - - - -+- - - - - - - - - - - - - - -+
```

Grouping Sets

Thus far in this chapter, we've seen simple group-by expressions that we can use to aggregate on a set of columns with the values in those columns. However, sometimes we want something a bit more complete—an aggregation across multiple groups. We achieve this by using *grouping sets*. Grouping sets are a low-level tool for combining sets of aggregations together. They give you the ability to create arbitrary aggregation in their group-by statements.

Let's work through an example to gain a better understanding. Here, we would like to get the total quantity of all stock codes and customers. To do so, we'll use the following SQL expression:

```
// i n Scal a val df NoNul l = df Wi t hDat e. drop( ) df
NoNul l . creat eOrRepl aceTempVi ew( "df NoNul l ")

# i n Pyt hon df NoNul l = df Wi t hDat e. drop( ) df NoNul
l . creat eOrRepl aceTempVi ew( "df NoNul l ")

- - i n SQL
SELECT Cust omerI d, st ockCode, sum( Quant i t y) FROM df NoNul l
GROUP BY cust omerI d, st ockCode
ORDER BY Cust omerI d DESC, st ockCode DESC

+- - - - - - - - - -+- - - - - - - - -+- - - - - - - - - - - - -+ |Cust
omerI d|st ockCode|sum( Quant i t y) |
```

```
+----------+---------+-------------+
|   18287|   85173|        48|
|   18287|  85040A|         48|
|   18287|  85039B|        120| . . .
|   18287|   23269|        36|
+----------+---------+-------------+
```

You can do the exact same thing by using a grouping set:

```
- - i n SQL
SELECT Cust omerI d, st ockCode, sum( Quant i t y) FROM df NoNul l
GROUP BY cust omerI d, st ockCode GROUPI NG SETS( ( cust omerI d, st ockCode) )
ORDER BY Cust omerI d DESC, st ockCode DESC

+----------+---------+-------------+ |Cust
omerI d|st ockCode|sum( Quant i t y) |
+----------+---------+-------------+
|   18287|   85173|        48|
|   18287|  85040A|         48|
|   18287|  85039B|        120| . . .
|   18287|   23269|        36|
+----------+---------+-------------+
```

WARNING

Grouping sets depend on null values for aggregation levels. If you do not filter-out null values, you will get incorrect results. This applies to cubes, rollups, and grouping sets.

Simple enough, but what if you *also* want to include the total number of items, regardless of customer or stock code? With a conventional group-by statement, this would be impossible. But, it's simple with grouping sets: we simply specify that we would like to aggregate at that level, as well, in our grouping set. This is, effectively, the union of several different groupings together:

```
- - i n SQL
SELECT Cust omerI d, st ockCode, sum( Quant i t y) FROM df NoNul l
GROUP BY cust omerI d, st ockCode GROUPI NG SETS( ( cust omerI d, st ockCode) , ( ) )
ORDER BY Cust omerI d DESC, st ockCode DESC

+----------+---------+-------------+ |cust
omerI d|st ockCode|sum( Quant i t y) |
+----------+---------+-------------+
|   18287|   85173|        48|
|   18287|  85040A|         48|
|   18287|  85039B|        120| . . .
|   18287|   23269|        36|
```

```
+----------+---------+-------------+
```

The GROUPI NG SETS operator is only available in SQL. To perform the same in DataFrames, you use the rol l up and cube operators—which allow us to get the same results. Let's go through those.

Rollups

Thus far, we've been looking at explicit groupings. When we set our grouping keys of multiple columns, Spark looks at those as well as the actual combinations that are visible in the dataset. A rollup is a multidimensional aggregation that performs a variety of group-by style calculations for us.

Let's create a rollup that looks across time (with our new Dat e column) and space (with the Count ry column) and creates a new DataFrame that includes the grand total over all dates, the grand total for each date in the DataFrame, and the subtotal for each country on each date in the DataFrame:

```
val rol l edUpDF = df NoNul l . rol l up( "Dat e", "Count ry") . agg( sum( "Quant i t y") )
  . sel ect Expr( "Dat e", "Count ry", "` sum( Quant i t y) ` as t ot al _quant i t y")
  . orderBy( "Dat e")
rol l edUpDF. show( )

# i n Pyt hon rol l edUpDF = df NoNul l . rol l up( "Dat e", "Count ry") . agg( sum( "Quant
i t y") ) \
  . sel ect Expr( "Dat e", "Count ry", "` sum( Quant i t y) ` as t ot al _quant i t y") \
  . orderBy( "Dat e") rol l
edUpDF. show( )

+----------+--------------+--------------
+ |     Dat e|      Count ry|t ot al _quant i t y|
+----------+--------------+--------------+
|     nul l |        nul l |      5176450|
|2010- 12- 01|Uni t ed Ki ngdom|       23949|
|2010- 12- 01|       Germany|         117|
|2010- 12- 01|        France|         449| . . .
|2010- 12- 03|        France|         239|
|2010- 12- 03|         I t al y|         164|
|2010- 12- 03|       Bel gi um|         528|
+----------+--------------+--------------+
```

Now where you see the nul l values is where you'll find the grand totals. A nul l in both rollup columns specifies the grand total across both of those columns: `rol l edUpDF. where( "Count ry I S NULL") . show( )`

```
rol l edUpDF. where( "Dat e I S NULL") . show( )

+- - - - +- - - - - - - +- - - - - - - - - - - - - - + |Dat
e|Count ry|t ot al _quant i t y|
+- - - - +- - - - - - - +- - - - - - - - - - - - - - + |nul
l |   nul l |      5176450|
+- - - - +- - - - - - - +- - - - - - - - - - - - - - +
```

Cube

A cube takes the rollup to a level deeper. Rather than treating elements hierarchically, a cube does the same thing across all dimensions. This means that it won't just go by date over the entire time period, but also the country. To pose this as a question again, can you make a table that includes the following?

- The total across all dates and countries
- The total for each date across all countries
- The total for each country on each date
- The total for each country across all dates

The method call is quite similar, but instead of calling rol l up, we call cube:

```
// i n Scal a df NoNul l . cube( "Dat e", "Count ry") . agg( sum( col ( "Quant
i t y") ) )
  . sel ect ( "Dat e", "Count ry", "sum( Quant i t y) ") . orderBy( "Dat e") . show( )

# i n Pyt hon f rom pyspark. sql . f unct i ons i
mport sum

df NoNul l . cube( "Dat e", "Count ry") . agg( sum( col ( "Quant i t y") ) ) \
  . sel ect ( "Dat e", "Count ry", "sum( Quant i t y) ") . orderBy( "Dat e") . show( )

+- - - - +- - - - - - - - - - - - - - - - - - - +- - - - - - - - - - - - - +
|Dat e|          Count ry|sum( Quant i t y) |
+- - - - +- - - - - - - - - - - - - - - - - - - +- - - - - - - - - - - - - +
|nul l |           J apan|       25218|
|nul l |         Port ugal |       16180|
|nul l |       Unspeci f i ed|        3300|
|nul l |            nul l |     5176450|
|nul l |         Aust ral i a|       83653| . . .
|nul l |           Norway|       19247|
|nul l |         Hong Kong|        4769|
|nul l |           Spai n|       26824|
|nul l |     Czech Republ i c|         592|
+- - - - +- - - - - - - - - - - - - - - - - - - +- - - - - - - - - - - - - +
```

This is a quick and easily accessible summary of nearly all of the information in our table, and it's a great way to create a quick summary table that others can use later on.

Grouping Metadata

Sometimes when using cubes and rollups, you want to be able to query the aggregation levels so that you can easily filter them down accordingly. We can do this by using the groupi ng_i d, which gives us a column specifying the level of aggregation that we have in our result set. The query in the example that follows returns four distinct grouping IDs:

Table 7-1. Purpose of grouping IDs

Grouping	Description ID
3	This will appear for the highest-level aggregation, which will gives us the total quantity regardless of cust omerI d and st ockCode.
2	This will appear for all aggregations of individual stock codes. This gives us the total quantity per stock code, regardless of customer.
1	This will give us the total quantity on a per-customer basis, regardless of item purchased.
0	This will give us the total quantity for individual cust omerI d and st ockCode combinations.

This is a bit abstract, so it's well worth trying out to understand the behavior yourself:

```
// i n Scal a i mport org. apache. spark. sql . f unct i ons. { groupi ng_i d, sum,
expr}

df NoNul l . cube( "cust omerI d", "st ockCode") . agg( groupi ng_i d( ) , sum( "Quant i t y") ) .
orderBy( expr( "groupi ng_i d( ) ") . desc)
. show( )

+- - - - - - - - - - +- - - - - - - - - +- - - - - - - - - - - - - +- - - - - - - - - - - - - +
|cust omerI d|st ockCode|groupi ng_i d( ) |sum( Quant i t y) |
+- - - - - - - - - - +- - - - - - - - - +- - - - - - - - - - - - - +- - - - - - - - - - - - - +
|    nul l |    nul l |         3|    5176450|
|    nul l |   23217|          2|       1309|
|    nul l |  90059E|          2|         19| . . .
+- - - - - - - - - - +- - - - - - - - - +- - - - - - - - - - - - - +- - - - - - - - - - - - - +
```

Pivot

Pivots make it possible for you to convert a row into a column. For example, in our current data we have a Count ry column. With a pivot, we can aggregate according to some function for each of those given countries and display them in an easy-to-query way:

```
// in Scala val pivoted = dfWithDate.groupBy("date").pivot("Country")
.sum()

# in Python
pivoted = dfWithDate.groupBy("date").pivot("Country").sum()
```

This DataFrame will now have a column for every combination of country, numeric variable, and a column specifying the date. For example, for USA we have the following columns: USA_sum(Quantity), USA_sum(UnitPrice), USA_sum(CustomerID). This represents one for each numeric column in our dataset (because we just performed an aggregation over all of them).

Here's an example query and result from this data: `pivoted.where("date > '2011-12-05'")`

```
.select("date", "`USA_sum(Quantity)`").show()

+----------+-----------------+
|      date|USA_sum(Quantity)|
+----------+-----------------+
|2011-12-06|             null|
|2011-12-09|             null|
|2011-12-08|             -196|
|2011-12-07|             null|
+----------+-----------------+
```

Now all of the columns can be calculated with single groupings, but the value of a pivot comes down to how you would like to explore the data. It can be useful, if you have low enough cardinality in a certain column to transform it into columns so that users can see the schema and immediately know what to query for.

User-Defined Aggregation Functions

User-defined aggregation functions (UDAFs) are a way for users to define their own aggregation functions based on custom formulae or business rules. You can use UDAFs to compute custom calculations over groups of input data (as opposed to single rows). Spark maintains a single AggregationBuffer to store intermediate results for every group of input data.

To create a UDAF, you must inherit from the UserDefinedAggregateFunction base class and implement the following methods:

- inputSchema represents input arguments as a StructType buffer
- erSchema represents intermediate UDAF results as a StructType dat
- aType represents the return DataType
-

det ermi ni st i c is a Boolean value that specifies whether this UDAF will return the same result for a given input i ni t i al i ze allows you to initialize values of an

- aggregation buffer updat e describes how you should update the internal buffer
- based on a given row merge describes how two aggregation buffers should be
- merged eval uat e will generate the final result of the aggregation
- following example implements a Bool And, which will inform us whether all the rows

The (for a given column) are true; if they're not, it will return false:

```
// i n Scal a i mport org. apache. spark. sql . expressi ons. Mut abl eAggregat i onBuf f
er i mport org. apache. spark. sql . expressi ons. UserDef i nedAggregat eFunct i on i
mport org. apache. spark. sql . Row i mport org. apache. spark. sql . t ypes. _ cl ass Bool
And ext ends UserDef i nedAggregat eFunct i on {   def i nput Schema: org. apache.
spark. sql . t ypes. St ruct Type =    St ruct Type( St ruct Fi el d( "val ue", Bool eanType)
: : Ni l )   def buf f erSchema: St ruct Type = St ruct Type(
   St ruct Fi el d( "resul t ", Bool eanType) : : Ni l
 )   def dat aType: Dat aType = Bool eanType   def det ermi ni st i c: Bool ean
= t rue   def i ni t i al i ze( buf f er: Mut abl eAggregat i onBuf f er) : Uni t = {
buf f er( 0) = t rue
 }
 def updat e( buf f er: Mut abl eAggregat i onBuf f er, i nput : Row) : Uni t = {    buf f er( 0)
= buf f er. get As[ Bool ean] ( 0) && i nput . get As[ Bool ean] ( 0)
 }   def merge( buf f er1: Mut abl eAggregat i onBuf f er, buf f er2: Row) : Uni t = {    buf f
er1( 0) = buf f er1. get As[ Bool ean] ( 0) && buf f er2. get As[ Bool ean] ( 0)
 }  def eval uat e( buf f er: Row) : Any = {    buf
f er( 0)
 }
}
```

Now, we simply instantiate our class and/or register it as a function:

```
// i n Scal a val ba = new Bool And spark. udf . regi
st er( "bool and", ba) i mport org. apache. spark.
sql . f unct i ons. _ spark. range( 1)
 . sel ect Expr( "expl ode( array( TRUE, TRUE, TRUE) ) as t ")
 . sel ect Expr( "expl ode( array( TRUE, FALSE, TRUE) ) as f ", "t ")
 . sel ect ( ba( col ( "t ") ) , expr( "bool and( f ) ") )
 . show( )

+- - - - - - - - - -+- - - - - - - - - -+ |bool
and( t ) |bool and( f ) |
+- - - - - - - - - -+- - - - - - - - - -+
|    t rue|    f al se|
+- - - - - - - - - -+- - - - - - - - - -+
```

UDAFs are currently available only in Scala or Java. However, in Spark 2.3, you will also be able to call Scala or Java UDFs and UDAFs by registering the function just as we showed in the UDF section in Chapter 6. For more information, go to SPARK-19439.

Conclusion

This chapter walked through the different types and kinds of aggregations that you can perform in Spark. You learned about simple grouping-to window functions as well as rollups and cubes.
Chapter 8 discusses how to perform joins to combine different data sources together.

Chapter 8. Joins

Chapter 7 covered aggregating single datasets, which is helpful, but more often than not, your Spark applications are going to bring together a large number of different datasets. For this reason, joins are an essential part of nearly all Spark workloads. Spark's ability to talk to different data means that you gain the ability to tap into a variety of data sources across your company. This chapter covers not just what joins exist in Spark and how to use them, but some of the basic internals so that you can think about how Spark actually goes about executing the join on the cluster. This basic knowledge can help you avoid running out of memory and tackle problems that you could not solve before.

Join Expressions

A *join* brings together two sets of data, the *left* and the *right*, by comparing the value of one or more *keys* of the left and right and evaluating the result of a *join expression* that determines whether Spark should bring together the left set of data with the right set of data. The most common join expression, an equi - j oi n, compares whether the specified keys in your left and right datasets are equal. If they are equal, Spark will combine the left and right datasets. The opposite is true for keys that do not match; Spark discards the rows that do not have matching keys. Spark also allows for much more sophsticated join policies in addition to equi-joins. We can even use complex types and perform something like checking whether a key exists within an array when you perform a join.

Join Types

Whereas the join expression determines whether two rows *should* join, the join type determines *what* should be in the result set. There are a variety of different join types available in Spark for you to use:

- Inner joins (keep rows with keys that exist in the left and right datasets)
- Outer joins (keep rows with keys in either the left or right datasets)
- Left outer joins (keep rows with keys in the left dataset)
- Right outer joins (keep rows with keys in the right dataset)
- Left semi joins (keep the rows in the left, and only the left, dataset where the key appears in the right dataset)
- Left anti joins (keep the rows in the left, and only the left, dataset where they do not appear in the right dataset)
- Natural joins (perform a join by implicitly matching the columns between the two datasets with the same names)
- Cross (or Cartesian) joins (match every row in the left dataset with every row in the right dataset)

If you have ever interacted with a relational database system, or even an Excel spreadsheet, the concept of joining different datasets together should not be too abstract. Let's move on to showing examples of each join type. This will make it easy to understand exactly how you can apply these to your own problems. To do this, let's create some simple datasets that we can use in our examples:

```
// i n Scal a val
person = Seq(
  ( 0, "Bi l l Chambers", 0, Seq( 100) ) ,
  ( 1, "Mat ei Zahari a", 1, Seq( 500, 250, 100) ) ,
  ( 2, "Mi chael Armbrust ", 1, Seq( 250, 100) ) )
 . t oDF( "i d", "name", "graduat e_program", "spark_st at us") val graduat
eProgram = Seq(
  ( 0, "Mast ers", "School of I nf ormat i on", "UC Berkel ey") ,
  ( 2, "Mast ers", "EECS", "UC Berkel ey") ,
  ( 1, "Ph. D. ", "EECS", "UC Berkel ey") )  . t oDF( "i d",
"degree", "depart ment ", "school ") val sparkSt at us = Seq(
( 500, "Vi ce Presi dent ") ,
  ( 250, "PMC Member") ,
  ( 100, "Cont ri but or") )
 . t oDF( "i d", "st at us")

# i n Pyt hon person = spark. creat eDat
aFrame( [
  ( 0, "Bi l l Chambers", 0, [ 100] ) ,
  ( 1, "Mat ei Zahari a", 1, [ 500, 250, 100] ) ,
  ( 2, "Mi chael Armbrust ", 1, [ 250, 100] ) ] ) \
 . t oDF( "i d", "name", "graduat e_program", "spark_st at us") graduat eProgram
= spark. creat eDat aFrame( [
```

```
  ( 0, "Mast ers", "School of I nf ormat i on", "UC Berkel ey") ,
  ( 2, "Mast ers", "EECS", "UC Berkel ey") ,
  ( 1, "Ph. D. ", "EECS", "UC Berkel ey") ] ) \   . t oDF( "i d",
"degree", "depart ment ", "school ") sparkSt at us = spark.
creat eDat aFrame( [
  ( 500, "Vi ce Presi dent ") ,
  ( 250, "PMC Member") ,
  ( 100, "Cont ri but or") ] ) \
 . t oDF( "i d", "st at us")
```

Next, let's register these as tables so that we use them throughout the chapter:

```
person. creat eOrRepl aceTempVi ew( "person") graduat eProgram. creat
eOrRepl aceTempVi ew( "graduat eProgram") sparkSt at us. creat eOrRepl
aceTempVi ew( "sparkSt at us")
```

Inner Joins

Inner joins evaluate the keys in both of the DataFrames or tables and include (and join together) only the rows that evaluate to true. In the following example, we join the graduat eProgram DataFrame with the person DataFrame to create a new DataFrame:

```
// i n Scal a val j oi nExpressi on = person. col ( "graduat e_program") === graduat eProgram. col ( "i d")

# i n Pyt hon
j oi nExpressi on = person[ "graduat e_program"] == graduat eProgram[ ' i d' ]
```

Keys that do not exist in both DataFrames will not show in the resulting DataFrame. For example, the following expression would result in zero values in the resulting DataFrame:

```
// i n Scal a val wrongJ oi nExpressi on = person. col ( "name") === graduat eProgram. col ( "school
")

# i n Pyt hon
wrongJ oi nExpressi on = person[ "name"] == graduat eProgram[ "school "]
```

Inner joins are the default join, so we just need to specify our left DataFrame and join the right in the JOIN expression: `person. j oi n( graduat eProgram, j oi nExpressi on) . show( )`

```
- - i n SQL
SELECT * FROM person J OI N graduat eProgram
  ON person. graduat e_program = graduat eProgram. i d
+- - - +- - - - - - - - - - - - - - - +- - - - - - - - - - - - - - - +- - - +- - - - - - - +- - - - - - - - - - +-
- -
| i d|         name|graduat e_program|   spark_st at us| i d| degree|depart ment |. . . +- - - +- - - - - - -
- - - - - - - - +- - - - - - - - - - - - - - - +- - - - - - - - - - - - - - - +- - - +- - - - - - - +- - - - - - - - - - +- - -
|  0|   Bi l l Chambers|            0|        [ 100] |  0|Mast ers| School . . . |. . .
|  1|   Mat ei Zahari a|            1|[ 500, 250, 100] |  1|  Ph. D. |      EECS|. . .
```

```
|  2|Mi chael Armbrust |            1|    [ 250, 100] |  1|  Ph. D. |     EECS|. . .
+- - - +- - - - - - - - - - - - - - - +- - - - - - - - - - - - - - - +- - - - - - - - - - - - - - - +- - - +- - - - - - - +- - - - - - - - - - +-
```

- We can also specify this explicitly by passing in a third parameter, the j oi nType:

```
// i n Scal a var j oi nType =
"i nner"

# i n Pyt hon j oi nType
= "i nner"

person. j oi n( graduat eProgram, j oi nExpressi on, j oi nType) . show( )

- - i n SQL
SELECT * FROM person I NNER J OI N graduat eProgram
  ON person. graduat e_program = graduat eProgram. i d

+- - - +- - - - - - - - - - - - - - - +- - - - - - - - - - - - - - - +- - - - - - - - - - - - - - - +- - - +- - - - - - - +- - - - - - - - - - - -
- -
| i d|         name|graduat e_program|   spark_st at us| i d| degree| depart ment . . . +- - - +- - - - - - - -
- - - - - - - - +- - - - - - - - - - - - - - - +- - - - - - - - - - - - - - - +- - - +- - - - - - - +- - - - - - - - - - - - -
|  0|   Bi l l Chambers|            0|        [ 100] |  0|Mast ers|     School . . .
|  1|   Mat ei Zahari a|            1|[ 500, 250, 100] |  1|  Ph. D. |      EECS. . .
|  2|Mi chael Armbrust |            1|    [ 250, 100] |  1|  Ph. D. |      EECS. . .
+- - - +- - - - - - - - - - - - - - - +- - - - - - - - - - - - - - - +- - - - - - - - - - - - - - - +- - - +- - - - - - - +- - - - - - - - - - - -
- -
```

Outer Joins

Outer joins evaluate the keys in both of the DataFrames or tables and includes (and joins together) the rows that evaluate to true or false. If there is no equivalent row in either the left or right DataFrame, Spark will insert nul l : j oi nType = "out er"

```
person. j oi n( graduat eProgram, j oi nExpressi on, j oi nType) . show( )

- - i n SQL
SELECT * FROM person FULL OUTER J OI N graduat eProgram
  ON graduat e_program = graduat eProgram. i d

+- - - - +- - - - - - - - - - - - - - - +- - - - - - - - - - - - - - - +- - - - - - - - - - - - - - - +- - - +- - - - - - - +- - - - - - - - - - - -
- -
| i d|         name|graduat e_program|   spark_st at us| i d| degree| depart men. . . +- - - - +- - - - - - - -
- - - - - - - - +- - - - - - - - - - - - - - - +- - - - - - - - - - - - - - - +- - - +- - - - - - - +- - - - - - - - - - - - |  1|  Mat ei
Zahari a|            1|[ 500, 250, 100] |  1|  Ph. D. |      EEC. . . |  2|Mi chael Armbrust |            1|    [
250, 100] |  1|  Ph. D. |      EEC. . .
|nul l |         nul l |         nul l |         nul l |  2|Mast ers|      EEC. . .
|  0|   Bi l l Chambers|            0|        [ 100] |  0|Mast ers|     School . . .
```

```
+----+---------------+---------------+---------------+---+-------+----------
--
```

Left Outer Joins

Left outer joins evaluate the keys in both of the DataFrames or tables and includes all rows from the left DataFrame as well as any rows in the right DataFrame that have a match in the left DataFrame. If there is no equivalent row in the right DataFrame, Spark will insert nul l : j oi nType = "l ef t _out er"

```
graduat eProgram. j oi n( person, j oi nExpressi on, j oi nType) . show( )

--i n SQL
SELECT * FROM graduat eProgram LEFT OUTER J OI N person
  ON person. graduat e_program = graduat eProgram. i d
```

```
+---+-------+----------+-----------+----+---------------+---------------+---
| i d| degree|depart ment |    school |  i d|         name|graduat e_program|. . . +---+-------+--
--------+-----------+----+---------------+---------------+---
|  0|Mast ers| School . . . |UC Berkel ey|   0|   Bi l l Chambers|           0|. . .
|  2|Mast ers|     EECS|UC Berkel ey|nul l |          nul l |          nul l |. . .
|  1|  Ph. D. |     EECS|UC Berkel ey|   2|Mi chael Armbrust |           1|. . .
|  1|  Ph. D. |     EECS|UC Berkel ey|   1|   Mat ei Zahari a|           1|. . .
+---+-------+----------+-----------+----+---------------+---------------+---
```

Right Outer Joins

Right outer joins evaluate the keys in both of the DataFrames or tables and includes all rows from the right DataFrame as well as any rows in the left DataFrame that have a match in the right DataFrame. If there is no equivalent row in the left DataFrame, Spark will insert nul l : j oi nType = "ri ght _out er"

```
person. j oi n( graduat eProgram, j oi nExpressi on, j oi nType) . show( )

--i n SQL
SELECT * FROM person RI GHT OUTER J OI N graduat eProgram
  ON person. graduat e_program = graduat eProgram. i d
```

```
+----+---------------+---------------+---------------+---+-------+----------
-+|  i d|         name|graduat e_program|   spark_st at us| i d| degree|  depart ment |
+----+---------------+---------------+---------------+---+-------+----------
-+
```

```
|  0|  Bi l l Chambers|           0|       [ 100] |  0|Mast ers|School of . . . |
|nul l |         nul l |         nul l |        nul l |  2|Mast ers|      EECS|
|  2|Mi chael Armbrust |           1|    [ 250, 100] |  1|  Ph. D. |      EECS|
|  1|  Mat ei Zahari a|           1|[ 500, 250, 100] |  1|  Ph. D. |      EECS|
+----+----------------+----------------+---------------+---+--------+----------
- +
```

Left Semi Joins

Semi joins are a bit of a departure from the other joins. They do not actually include any values from the right DataFrame. They only compare values to see if the value exists in the second DataFrame. If the value does exist, those rows will be kept in the result, even if there are duplicate keys in the left DataFrame. Think of left semi joins as filters on a DataFrame, as opposed to the function of a conventional join: j oi nType = "l ef t _semi "

```
graduat eProgram. j oi n( person, j oi nExpressi on, j oi nType) . show( )

+---+--------+--------------------+-----------
+ | i d| degree|        depart ment |     school |
+---+--------+--------------------+-----------+
|  0|Mast ers|School of I nf orma. . . |UC Berkel ey|
|  1|  Ph. D. |              EECS|UC Berkel ey|
+---+--------+--------------------+-----------+

// i n Scal a val gradProgram2 = graduat eProgram. uni
on( Seq(
   ( 0, "Mast ers", "Dupl i cat ed Row", "Dupl i cat ed School ") ) . t oDF( ) ) gradProgram2. creat

eOrRepl aceTempVi ew( "gradProgram2")

# i n Pyt hon gradProgram2 = graduat eProgram. uni on( spark. creat eDat
aFrame( [    ( 0, "Mast ers", "Dupl i cat ed Row", "Dupl i cat ed School ") ] ) )
gradProgram2. creat eOrRepl aceTempVi ew( "gradProgram2")

gradProgram2. j oi n( person, j oi nExpressi on, j oi nType) . show( )

- - i n SQL
SELECT * FROM gradProgram2 LEFT SEMI J OI N person
 ON gradProgram2. i d = person. graduat e_program

+---+--------+--------------------+-----------------
+ | i d| degree|        depart ment |          school |
+---+--------+--------------------+-----------------+
|  0|Mast ers|School of I nf orma. . . |      UC Berkel ey|
|  1|  Ph. D. |              EECS|      UC Berkel ey|
|  0|Mast ers|     Dupl i cat ed Row|Dupl i cat ed School |
```

```
+---+-------+--------------------+----------------+
```

Left Anti Joins

Left anti joins are the opposite of left semi joins. Like left semi joins, they do not actually include any values from the right DataFrame. They only compare values to see if the value exists in the second DataFrame. However, rather than keeping the values that exist in the second
DataFrame, they keep only the values that *do not* have a corresponding key in the second DataFrame. Think of anti joins as a NOT I N SQL-style filter:

```
j oi nType = "l ef t _ant i " graduat eProgram. j oi n( person, j oi nExpressi on,
j oi nType) . show( )

- - i n SQL
SELECT * FROM graduat eProgram LEFT ANTI J OI N person
  ON graduat eProgram. i d = person. graduat e_program
```

```
+---+-------+----------+-----------+
| i d| degree|depart ment |    school |
+---+-------+----------+-----------+
|  2|Mast ers|     EECS|UC Berkel ey|
+---+-------+----------+-----------+
```

Natural Joins

Natural joins make implicit guesses at the columns on which you would like to join. It finds matching columns and returns the results. Left, right, and outer natural joins are all supported.

> **WARNING**
>
> Implicit is always dangerous! The following query will give us incorrect results because the two DataFrames/tables share a column name (i d), but it means different things in the datasets. You should always use this join with caution.
>
> ```
> - - i n SQL
> SELECT * FROM graduat eProgram NATURAL J OI N person
> ```

Cross (Cartesian) Joins

The last of our joins are cross-joins or *cartesian products*. Cross-joins in simplest terms are inner joins that do not specify a predicate. Cross joins will join every single row in the left DataFrame to ever single row in the right DataFrame. This will cause an absolute explosion in the number of rows contained in the resulting DataFrame. If you have 1,000 rows in each DataFrame, the crossjoin of these will result in 1,000,000 (1,000 x 1,000) rows. For this reason, you must very explicitly state that you want a cross-join by using the cross join keyword:

```
j oi nType = "cross" graduat eProgram. j oi n( person, j oi nExpressi on, j oi
nType) . show( )

- - i n SQL
SELECT * FROM graduat eProgram CROSS J OI N person
  ON graduat eProgram. i d = person. graduat e_program
```

```
+---+-------+----------+-----------+---+---------------+----------------+-----
--
| id| degree|department |    school | id|         name|graduate_program|spar...+---+-------+-
---------+-----------+---+---------------+----------------+-------
|  0|Masters| School...|UC Berkeley|  0|  Bill Chambers|               0|   ...
|  1|  Ph.D.|     EECS|UC Berkeley|  2|Michael Armbrust|               1|  [2...
|  1|  Ph.D.|     EECS|UC Berkeley|  1|  Matei Zaharia|               1|[500...
+---+-------+----------+-----------+---+---------------+----------------+-----
```

- If you truly intend to have a cross-join, you can call that out explicitly: person. crossJoin(

graduateProgram) . show( )

```
--in SQL
SELECT * FROM graduateProgram CROSS JOIN person
```

```
+---+---------------+---------------+---------------+---+-------+-----------
-+| id|         name|graduate_program|  spark_status| id| degree|  department...|
+---+---------------+---------------+---------------+---+-------+-----------
-+|  0|  Bill Chambers|              0|         [100]|  0|Masters|   School...|...
|  1|  Matei Zaharia|              1|[500, 250, 100]|  0|Masters|   School...|...
|  2|Michael Armbrust|              1|    [250, 100]|  0|Masters|   School...|...
+---+---------------+---------------+---------------+---+-------+-----------
-+
```

WARNING

You should use cross-joins only if you are absolutely, 100 percent sure that this is the join you need. There is a reason why you need to be explicit when defining a cross-join in Spark. They're dangerous! Advanced users can set the session-level configuration spark.sql.crossJoin.enable to true in order to allow cross-joins without warnings or without Spark trying to perform another join for you.

Challenges When Using Joins

When performing joins, there are some specific challenges and some common questions that arise. The rest of the chapter will provide answers to these common questions and then explain how, at a high level, Spark performs joins. This will hint at some of the optimizations that we are going to cover in later parts of this book.

Joins on Complex Types

Even though this might seem like a challenge, it's actually not. Any expression is a valid join expression, assuming that it returns a Boolean:

```
import org.apache.spark.sql.functions.expr
```

```
person. wi t hCol umnRenamed( "i d", "personI d")
  . j oi n( sparkSt at us, expr( "array_cont ai ns( spark_st at us, i d) ") ) . show( )

# i n Pyt hon f rom pyspark. sql . f unct i ons i
mport expr

person. wi t hCol umnRenamed( "i d", "personI d") \
  . j oi n( sparkSt at us, expr( "array_cont ai ns( spark_st at us, i d) ") ) . show( )

- - i n SQL
SELECT * FROM
  ( sel ect i d as personI d, name, graduat e_program, spark_st at us FROM person)
  I NNER J OI N sparkSt at us ON array_cont ai ns( spark_st at us, i d)
```

```
+- - - - - - - -+- - - - - - - - - - - - - - - +- - - - - - - - - - - - - - - +- - - - - - - - - - - - - - +- - - +- - - - - - - - - - - - -
+ |personI d|         name|graduat e_program|   spark_st at us| i d|       st at us|
+- - - - - - - -+- - - - - - - - - - - - - - - +- - - - - - - - - - - - - - - +- - - - - - - - - - - - - - +- - - +- - - - - - - - - - - - -
+
|     0|   Bi l l Chambers|            0|        [ 100] |100|   Cont ri but or|
|     1|   Mat ei Zahari a|            1|[ 500, 250, 100] |500|Vi ce Presi dent |
|     1|   Mat ei Zahari a|            1|[ 500, 250, 100] |250|    PMC Member|
|     1|   Mat ei Zahari a|            1|[ 500, 250, 100] |100|   Cont ri but or|
|     2|Mi chael Armbrust |            1|    [ 250, 100] |250|    PMC Member|
|     2|Mi chael Armbrust |            1|    [ 250, 100] |100|   Cont ri but or|
+- - - - - - - -+- - - - - - - - - - - - - - - +- - - - - - - - - - - - - - - +- - - - - - - - - - - - - - +- - - +- - - - - - - - - - - - -
+
```

Handling Duplicate Column Names

One of the tricky things that come up in joins is dealing with duplicate column names in your results DataFrame. In a DataFrame, each column has a unique ID within Spark's SQL Engine, Catalyst. This unique ID is purely internal and not something that you can directly reference. This makes it quite difficult to refer to a specific column when you have a DataFrame with duplicate column names.

This can occur in two distinct situations:

- The join expression that you specify does not remove one key from one of the input DataFrames and the keys have the same column name
- Two columns on which you are not performing the join have the same name

Let's create a problem dataset that we can use to illustrate these problems: val

```
gradProgramDupe = graduat eProgram. wi t hCol umnRenamed( "i d", "graduat e_program")

val j oi nExpr = gradProgramDupe. col ( "graduat e_program") === person. col (    "graduat
e_program")
```

Note that there are now two graduat e_program columns, even though we joined on that key:

```
person. j oi n( gradProgramDupe, j oi nExpr) . show( )
```

The challenge arises when we refer to one of these columns: person. j oi n(

```
gradProgramDupe, j oi nExpr) . sel ect ( "graduat e_program") . show( )
```

Given the previous code snippet, we will receive an error. In this particular example, Spark generates this message:

```
org. apache. spark. sql . Anal ysi sExcept i on: Ref erence ' graduat e_program' i s ambi guous,
coul d be: graduat e_program#40, graduat e_program#1079. ;
```

Approach 1: Different join expression

When you have two keys that have the same name, probably the easiest fix is to change the join expression from a Boolean expression to a string or sequence. This automatically removes one of the columns for you during the join: person. j oi n(gradProgramDupe, "graduat e_program") . sel ect ("graduat e_program") . show( )

Approach 2: Dropping the column after the join

Another approach is to drop the offending column after the join. When doing this, we need to refer to the column via the original source DataFrame. We can do this if the join uses the same key names or if the source DataFrames have columns that simply have the same name:

```
person. j oi n( gradProgramDupe, j oi nExpr) . drop( person. col ( "graduat e_program") )
  . sel ect ( "graduat e_program") . show( )

val j oi nExpr = person. col ( "graduat e_program") === graduat eProgram. col ( "i d") person. j oi n(
graduat eProgram, j oi nExpr) . drop( graduat eProgram. col ( "i d") ) . show( )
```

This is an artifact of Spark's SQL analysis process in which an explicitly referenced column will pass analysis because Spark has no need to resolve the column. Notice how the column uses the
. col method instead of a col umn function. That allows us to implicitly specify that column by its specific ID.

Approach 3: Renaming a column before the join

We can avoid this issue altogether if we rename one of our columns before the join:

```
val gradProgram3 = graduat eProgram. wi t hCol umnRenamed( "i d", "grad_i d") val j oi nExpr =
person. col ( "graduat e_program") === gradProgram3. col ( "grad_i d") person. j oi n(
gradProgram3, j oi nExpr) . show( )
```

How Spark Performs Joins

To understand how Spark performs joins, you need to understand the two core resources at play: the *node-to-node communication strategy* and *per node computation strategy*. These internals are likely irrelevant to your business problem. However, comprehending how Spark performs joins can mean the difference between a job that completes quickly and one that never completes at all.

Communication Strategies

Spark approaches cluster communication in two different ways during joins. It either incurs a *shuffle join*, which results in an all-to-all communication or a *broadcast join*. Keep in mind that there is a lot more detail than we're letting on at this point, and that's intentional. Some of these internal optimizations are likely to change over time with new improvements to the cost-based optimizer and improved communication strategies. For this reason, we're going to focus on the high-level examples to help you understand exactly what's going on in some of the more common scenarios, and let you take advantage of some of the low-hanging fruit that you can use right away to try to speed up some of your workloads.

The core foundation of our simplified view of joins is that in Spark you will have either a big table or a small table. Although this is obviously a spectrum (and things do happen differently if you have a "medium-sized table"), it can help to be binary about the distinction for the sake of this explanation.

Big table–to–big table

When you join a big table to another big table, you end up with a shuffle join, such as that illustrates in Figure 8-1.

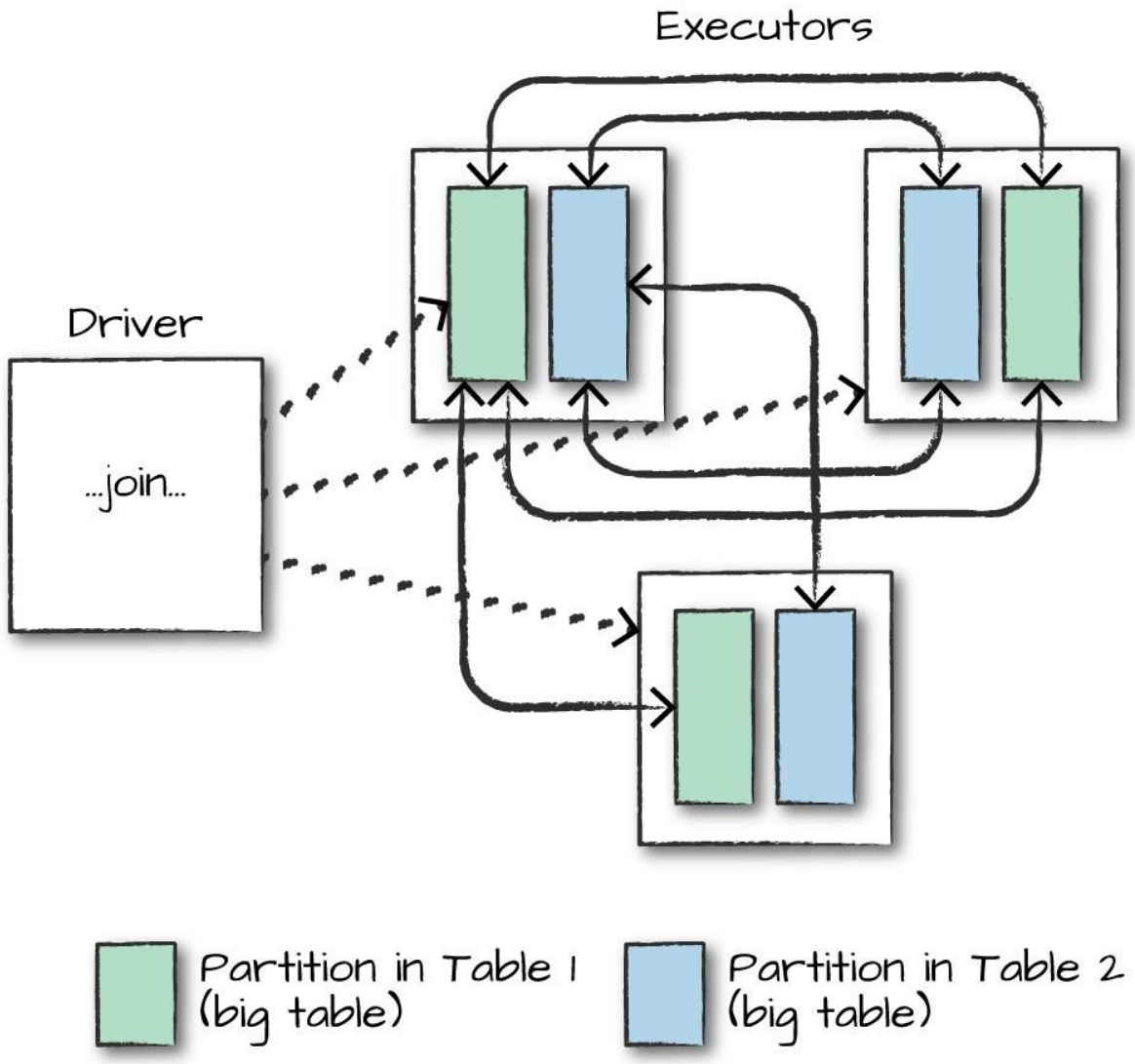


Figure 8-1. Joining two big tables

In a shuffle join, every node talks to every other node and they share data according to which node has a certain key or set of keys (on which you are joining). These joins are expensive because the network can become congested with traffic, especially if your data is not partitioned well.

This join describes taking a big table of data and joining it to another big table of data. An example of this might be a company that receives billions of messages every day from the Internet of Things, and needs to identify the day-over-day changes that have occurred. The way to do this is by joining on devi ceI d, messageType, and dat e in one column, and dat e - 1 day in the other column.

In Figure 8-1, DataFrame 1 and DataFrame 2 are both large DataFrames. This means that all worker nodes (and potentially every partition) will need to communicate with one another during the *entire* join process (with no intelligent partitioning of data).

Big table–to–small table

When the table is small enough to fit into the memory of a single worker node, with some breathing room of course, we can optimize our join. Although we can use a big table–to–big table communication strategy, it can often be more efficient to use a broadcast join. What this means is that we will replicate our small DataFrame onto every worker node in the cluster (be it located on one machine or many). Now this sounds expensive. However, what this does is prevent us from performing the all-to-all communication during the *entire* join process. Instead, we perform it only once at the beginning and then let each individual worker node perform the work without having to wait or communicate with any other worker node, as is depicted in Figure 8-2.

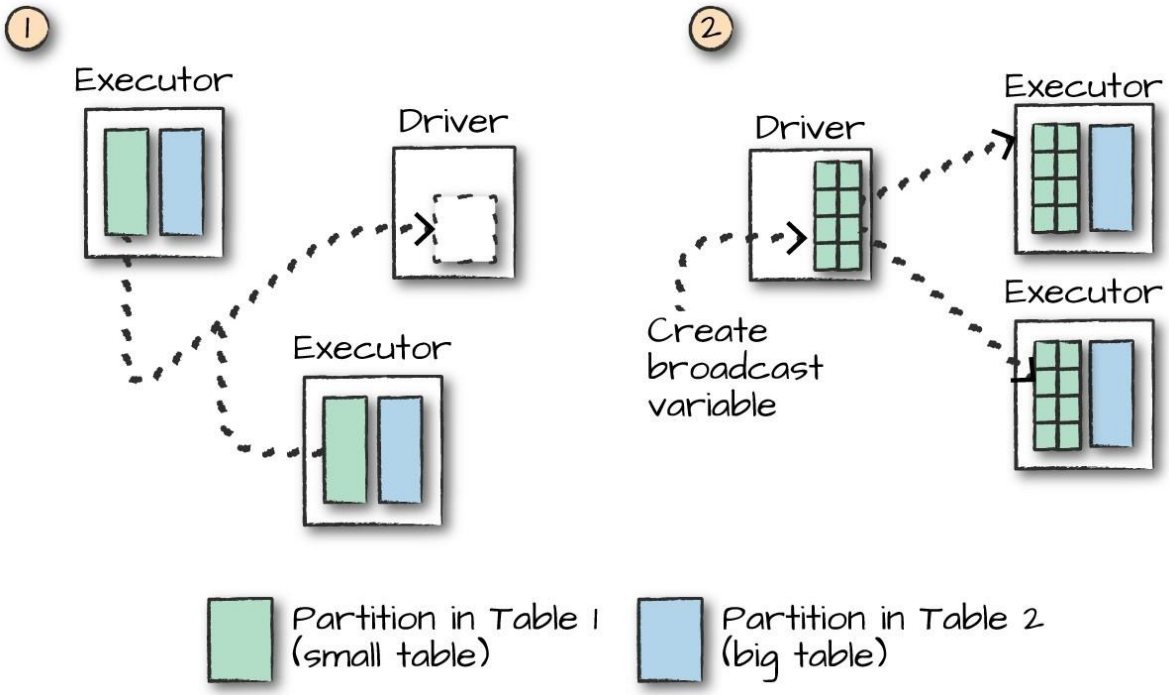


Figure 8-2. A broadcast join

At the beginning of this join will be a large communication, just like in the previous type of join. However, immediately after that first, there will be no further communication between nodes. This means that joins will be performed on every single node individually, making CPU the biggest bottleneck. For our current set of data, we can see that Spark has automatically set this up as a broadcast join by looking at the explain plan:

```
val j oi nExpr = person. col ( "graduat e_program") === graduat eProgram. col ( "i d") person. j oi n(

graduat eProgram, j oi nExpr) . expl ai n( )

== Physi cal Pl an ==
*Broadcast HashJ oi n [ graduat e_program#40] , [ i d#5. . . .
: - Local Tabl eScan [ i d#38, name#39, graduat e_progr. . .
+- Broadcast Exchange HashedRel at i onBroadcast Mode( . . . .
```

```
+- Local Tabl eScan [ i d#56, degree#57, depart men. . . .
```

With the DataFrame API, we can also explicitly give the optimizer a hint that we would like to use a broadcast join by using the correct function around the small DataFrame in question. In this example, these result in the same plan we just saw; however, this is not always the case:

```
i mport org. apache. spark. sql . f unct i ons. broadcast val j oi nExpr = person. col ( "graduat
e_program") === graduat eProgram. col ( "i d") person. j oi n( broadcast ( graduat eProgram) , j
oi nExpr) . expl ai n( )
```

The SQL interface also includes the ability to provide *hints* to perform joins. These are not *enforced*, however, so the optimizer might choose to ignore them. You can set one of these hints by using a special comment syntax. MAPJ OI N, BROADCAST, and BROADCASTJ OI N all do the same thing and are all supported:

```
- - i n SQL
SELECT /*+ MAPJ OI N( graduat eProgram) */ * FROM person J OI N graduat eProgram    ON
person. graduat e_program = graduat eProgram. i d
```

This doesn't come for free either: if you try to broadcast something too large, you can crash your driver node (because that collect is expensive). This is likely an area for optimization in the future.

Little table–to–little table

When performing joins with small tables, it's usually best to let Spark decide how to join them. You can always force a broadcast join if you're noticing strange behavior.

Conclusion

In this chapter, we discussed joins, probably one of the most common use cases. One thing we did not mention but is important to consider is if you partition your data correctly *prior to a join*, you can end up with much more efficient execution because even if a shuffle is planned, if data from two different DataFrames is already located on the same machine, Spark can avoid the shuffle. Experiment with some of your data and try partitioning beforehand to see if you can notice the increase in speed when performing those joins. In Chapter 9, we will discuss Spark's data source APIs. There are additional implications when you decide what order joins should occur in. Because some joins act as filters, this can be a low-hanging improvement in your workloads, as you are guaranteed to reduce data exchanged over the network.

The next chapter will depart from user manipulation, as we've seen in the last several chapters, and touch on reading and writing data using the Structured APIs.

Chapter 9. Data Sources

This chapter formally introduces the variety of other data sources that you can use with Spark out of the box as well as the countless other sources built by the greater community. Spark has six "core" data sources and hundreds of external data sources written by the community. The ability to read and write from all different kinds of data sources and for the community to create its own contributions is arguably one of Spark's greatest strengths. Following are Spark's core data sources:

- CSV
- JSON
- Parquet
- ORC
- JDBC/ODBC connections
- Plain-text files

As mentioned, Spark has numerous community-created data sources. Here's just a small sample:

- Cassandra
- HBase
- MongoDB
- AWS Redshift
- XML
- And many, many others

The goal of this chapter is to give you the ability to read and write from Spark's core data sources and know enough to understand what you should look for when integrating with thirdparty data sources. To achieve this, we will focus on the core concepts that you need to be able to recognize and understand.

The Structure of the Data Sources API

Before proceeding with how to read and write from certain formats, let's visit the overall organizational structure of the data source APIs.

Read API Structure

The core structure for reading data is as follows:

```
Dat aFrameReader. f ormat ( . . . ) . opt i on( "key", "val ue") . schema( . . . ) . l oad( )
```

We will use this format to read from all of our data sources. f ormat is optional because by default Spark will use the Parquet format. opt i on allows you to set key-value configurations to parameterize how you will read data. Lastly, schema is optional if the data source provides a schema or if you intend to use schema inference. Naturally, there are some required options for each format, which we will discuss when we look at each format.

> **NOTE**
>
> There is a lot of shorthand notation in the Spark community, and the data source read API is no exception. We try to be consistent throughout the book while still revealing some of the shorthand notation along the way.

Basics of Reading Data

The foundation for reading data in Spark is the Dat aFrameReader. We access this through the

SparkSessi on via the read attribute: spark. read

After we have a DataFrame reader, we specify several values:

- The *format*
- The *schema*
- The *read mode*
- A series of *options*

The format, options, and schema each return a Dat aFrameReader that can undergo further transformations and are all optional, except for one option. Each data source has a specific set of options that determine how the data is read into Spark (we cover these options shortly). At a minimum, you must supply the Dat aFrameReader a path to from which to read.

Here's an example of the overall layout:

```
spark. read. f ormat ( "csv")   . opt i
on( "mode", "FAI LFAST")
  . opt i on( "i nf erSchema", "t rue")
  . opt i on( "pat h", "pat h/t o/f i l e( s) ")
  . schema( someSchema)
  . l oad( )
```

There are a variety of ways in which you can set options; for example, you can build a map and pass in your configurations. For now, we'll stick to the simple and explicit way that you just saw.

Read modes

Reading data from an external source naturally entails encountering malformed data, especially when working with only semi-structured data sources. Read modes specify what will happen when Spark does come across malformed records. Table 9-1 lists the read modes.

Table 9-1. Spark's read modes

Read mode	Description
permi ssi ve	Sets all fields to nul l when it encounters a corrupted record and places all corrupted records in a string column called _corrupt _record
dropMal f ormed	Drops the row that contains malformed records
f ai l Fast	Fails immediately upon encountering malformed records

The default is permi ssi ve.

Write API Structure

The core structure for writing data is as follows:

```
Dat aFrameWri t er. f ormat ( . . . ) . opt i on( . . . ) . part i t i onBy( . . . ) . bucket By( . . . ) . sort By(  .
. . ) . save( )
```

We will use this format to write to all of our data sources. f ormat is optional because by default, Spark will use the arquet format. opt i on, again, allows us to configure how to write out our given data. Part i t i onBy, bucket By, and sort By work only for file-based data sources; you can use them to control the specific layout of files at the destination.

Basics of Writing Data

The foundation for writing data is quite similar to that of reading data. Instead of the Dat aFrameReader, we have the Dat aFrameWri t er. Because we always need to write out some given data source, we access the Dat aFrameWri t er on a per-Dat aFrame basis via the wri t e attribute:

```
// i n Scal a dat
aFrame. wri t e
```

After we have a Dat aFrameWri t er, we specify three values: the f ormat , a series of opt i ons, and the save mode. At a minimum, you must supply a path. We will cover the potential for options, which vary from data source to data source, shortly.

```
// i n Scal a dat af rame. wri t e. f
ormat ( "csv")
  . opt i on( "mode", "OVERWRI TE")
  . opt i on( "dat eFormat ", "yyyy- MM- dd")
  . opt i on( "pat h", "pat h/t o/f i l e( s) ")
  . save( )
```

Save modes

Save modes specify what will happen if Spark finds data at the specified location (assuming all else equal). Table 9-2 lists the save modes.

Table 9-2. Spark's save modes

Save mode	Description
append	Appends the output files to the list of files that already exist at that location
overwri t e	Will completely overwrite any data that already exists there
errorI f Exi st s	Throws an error and fails the write if data or files already exist at the specified location
i gnore	If data or files exist at the location, do nothing with the current DataFrame

The default is errorI f Exi st s. This means that if Spark finds data at the location to which you're writing, it will fail the write immediately.

We've largely covered the core concepts that you're going to need when using data sources, so now let's dive into each of Spark's native data sources.

CSV Files

CSV stands for commma-separated values. This is a common text file format in which each line represents a single record, and commas separate each field within a record. CSV files, while seeming well structured, are actually one of the trickiest file formats you will encounter because not many assumptions can be made in production scenarios about what they contain or how they are structured. For this reason, the CSV reader has a large number of options. These options give you the ability to work around issues like certain characters needing to be escaped—for example, commas inside of columns when the file is also comma-delimited or nul l values labeled in an unconventional way.

CSV Options

Table 9-3 presents the options available in the CSV reader.

Table 9-3. CSV data source options

Table 9-3. CSV data source options

Read/write	Key	Potential values	Default	Description
Both	sep	single string	,	The single character that is used as separator for each character field and value.
Both	header	true, false	false	A Boolean flag that declares whether the first line in the file(s) are the names of the columns.
Read	escape	string character	\	The character Spark should use to escape other characters in the file.
Read	inferSchema	true, false	false	Specifies whether Spark should infer column types when reading the file.
Read	ignoreLeadingWhiteSpace	true, false	false	Declares whether leading spaces from values being read should be skipped.
Read	ignoreTrailingWhiteSpace	true, false	false	Declares whether trailing spaces from values being read should be skipped.
Both	nullValue	Any string character	""	Declares what character represents a null value in the file.
Both	nanValue	Any string character	NaN	Declares what character represents a NaN or missing character in the CSV file.
Both	positiveInf	string or character value.	Inf	Declares what character(s) represent a positive infinite

Both	negat i veI nf	Any string or character	-Inf	Declares what character(s) represent a negative infinite value.
Both	compressi on or codec	None, uncompressed, bzi p2, def l at e, gzi p, l z4, or snappy	none	Declares what compression codec Spark should use to read or write the file.
Both	t i mest ampFormat	Any string or character that conforms to java's Si mpl eDat aFormat .	yyyy-MM-dd	Decla for an date t
Both	dat eFormat	Any string or character that conforms to java's Si mpl eDat aFormat .	yyyy-MM-dd'T'HH:mm:ss.SSSZZ	Declares the format for any columns that are timestamp type.
Read	maxCol umns	Any integer	20480	Declares the maximum number of columns in the file.
Read	maxCharsPerCol umn	Any integer	1000000	Declares the maximum number of characters in a column.
Read	escapeQuot es	t rue, f al se	t rue	Declares whether Spark should escape quotes that are found in lines.
Read	maxMal f ormedLogPerPart i t i on	Any integer	10	Sets the maximum number of malformed rows Spark will log for each partition. Malformed records beyond this number will be ignored.
Write	quot eAl l	t rue, f al se	f al se	Specifies whether all values should be enclosed in quotes, as opposed to just escaping values that have a quote character.
				This option allows you to read multiline CSV files

Read	mul t i Li ne	t rue, f al se	f al se	where each logical row in the CSV file might span multiple rows in the file itself.

Reading CSV Files

To read a CSV file, like any other format, we must first create a Dat aFrameReader for that specific format. Here, we specify the format to be CSV:

```
spark. read. f ormat ( "csv")
```

After this, we have the option of specifying a schema as well as modes as options. Let's set a couple of options, some that we saw from the beginning of the book and others that we haven't seen yet. We'll set the header to t rue for our CSV file, the mode to be FAI LFAST, and i nf erSchema to t rue:

```
// i n Scal a spark. read. f ormat (
"csv")    . opt i on( "header", "t
rue")
 . opt i on( "mode", "FAI LFAST")
 . opt i on( "i nf erSchema", "t rue")
 . l oad( "some/pat h/t o/f i l e. csv")
```

As mentioned, we can use the mode to specify how much tolerance we have for malformed data. For example, we can use these modes and the schema that we created in Chapter 5 to ensure that our file(s) conform to the data that we expected:

```
// i n Scal a i mport org. apache. spark. sql . t ypes. { St ruct Fi el d, St ruct Type, St ri ngType, LongType} val
myManual Schema = new St ruct Type( Array(   new St ruct Fi el d( "DEST_COUNTRY_NAME", St ri ngType,
t rue) ,   new St ruct Fi el d( "ORI GI N_COUNTRY_NAME", St ri ngType, t rue) ,   new St ruct Fi el d( "count
", LongType, f al se)
) ) spark. read. f ormat ( "csv")   .
opt i on( "header", "t rue")
 . opt i on( "mode", "FAI LFAST")
 . schema( myManual Schema)
 . l oad( "/dat a/f l i ght - dat a/csv/2010- summary. csv")   .
show( 5)
```

Things get tricky when we don't expect our data to be in a certain format, but it comes in that way, anyhow. For example, let's take our current schema and change all column types to LongType. This does not match the *actual* schema, but Spark has no problem with us doing this. The problem will only manifest itself when Spark actually reads the data. As soon as we start our Spark job, it will immediately fail (after we execute a job) due to the data not conforming to the specified schema:

```
// in Scala val myManualSchema = new StructType(Array(
new StructField("DEST_COUNTRY_NAME", LongType, true),
new StructField("ORIGIN_COUNTRY_NAME", LongType, true),
new StructField("count", LongType, false)))

spark.read.format("csv")
.option("header", "true")
.option("mode", "FAILFAST")
.schema(myManualSchema)
.load("/data/flight-data/csv/2010-summary.csv")
.take(5)
```

In general, Spark will fail only at job execution time rather than DataFrame definition time—even if, for example, we point to a file that does not exist. This is due to *lazy evaluation*, a concept we learned about in Chapter 2.

Writing CSV Files

Just as with reading data, there are a variety of options (listed in Table 9-3) for writing data when we write CSV files. This is a subset of the reading options because many do not apply when writing data (like maxColumns and inferSchema). Here's an example:

```
// in Scala val csvFile = spark.read.format(
"csv")
.option("header", "true").option("mode", "FAILFAST").schema(myManualSchema)
.load("/data/flight-data/csv/2010-summary.csv")

# in Python csvFile = spark.read.format(
"csv") \
.option("header", "true") \
.option("mode", "FAILFAST") \
.option("inferSchema", "true") \
.load("/data/flight-data/csv/2010-summary.csv")
```

For instance, we can take our CSV file and write it out as a TSV file quite easily:

```
// in Scala csvFile.write.format("csv").mode("overwrite").option("sep",
"\t")  .save("/tmp/my-tsv-file.tsv")

# in Python csvFile.write.format("csv").mode("overwrite").option("sep",
"\t") \
.save("/tmp/my-tsv-file.tsv")
```

When you list the destination directory, you can see that *my-tsv-file* is actually a folder with numerous files within it:

```
$ ls /tmp/my-tsv-file.tsv/
```

/t mp/my- t sv- f i l e. t sv/part - 00000- 35cf 9453- 1943- 4a8c- 9c82- 9f 6ea9742b29. csv

This actually reflects the number of partitions in our DataFrame at the time we write it out. If we were to repartition our data before then, we would end up with a different number of files. We discuss this trade-off at the end of this chapter.

JSON Files

Those coming from the world of JavaScript are likely familiar with JavaScript Object Notation, or JSON, as it's commonly called. There are some catches when working with this kind of data that are worth considering before we jump in. In Spark, when we refer to JSON files, we refer to *line-delimited* JSON files. This contrasts with files that have a large JSON object or array per file.

The line-delimited versus multiline trade-off is controlled by a single option: mul t i Li ne. When you set this option to t rue, you can read an entire file as one j son object and Spark will go through the work of parsing that into a DataFrame. Line-delimited JSON is actually a much more stable format because it allows you to append to a file with a new record (rather than having to read in an entire file and then write it out), which is what we recommend that you use. Another key reason for the popularity of line-delimited JSON is because JSON objects have structure, and JavaScript (on which JSON is based) has at least basic types. This makes it easier to work with because Spark can make more assumptions on our behalf about the data. You'll notice that there are significantly less options than we saw for CSV because of the objects.

JSON Options

Table 9-4 lists the options available for the JSON object, along with their descriptions.

Table 9-4. JSON data source options

Read/write	Key	Potential values	Default
D			
Both		None, uncompressed, compressi on or codec z4, or s snappy r	bzi p2, def l at e, none c gzi p, l D w c t
Both	dat eFormat	Any string or character that conforms to Java's Si mpl eDat aFormat .	yyyy-MM-dd
			D

Both	t i mest ampFormat	Any string or conforms to Java’s	t character that	f yyyy-MM-dd’T’HH:mm:ss.SSSZZ Si mpl eDat aFormat .		d f c a D a t t t
Read	pri mi t i veAsSt ri ng	t rue, f al se	f al se			
Read	al l owComment s	t rue, f al se	f al se			l p v s I J s c J r
Read	al l owUnquot edFi el dNames	t rue, f al se	f al se			
Read	al l owSi ngl eQuot es	t rue, f al se	t rue			A u J n A q a d q

Read	al l owNumeri cLeadi ngZeros	t rue, f al se	f al se			
c Read	al l owBacksl ashEscapi ngAnyCharact er	t rue, f al se	f al se			
Read	col umnNameOf Corrupt Record	Any string	Value of spark. sql . col umn&NameOf Corrupt Record			

Al zn(Aaqubqm

Arhmspmwtc n b

Read	mul t i Li ne	t rue, f al se	f al se	v A r n d J

Now, reading a line-delimited JSON file varies only in the format and the options that we specify: spark. read. f ormat ("j son")

Reading JSON Files

Let's look at an example of reading a JSON file and compare the options that we're seeing:

```
// i n Scal a spark. read. f ormat ( "j son") . opt i on( "mode", "FAI LFAST") . schema( myManual
Schema)   . l oad( "/dat a/f l i ght - dat a/j son/2010- summary. j son") . show( 5)

# i n Pyt hon spark. read. f ormat ( "j son") . opt i on( "mode", "FAI
LFAST") \
  . opt i on( "i nf erSchema", "t rue") \
  . l oad( "/dat a/f l i ght - dat a/j son/2010- summary. j son") . show( 5)
```

Writing JSON Files

Writing JSON files is just as simple as reading them, and, as you might expect, the data source does not matter. Therefore, we can reuse the CSV DataFrame that we created earlier to be the source for our JSON file. This, too, follows the rules that we specified before: one file per partition will be written out, and the entire DataFrame will be written out as a folder. It will also have one JSON object per line:

```
// i n Scal a csvFi l e. wri t e. f ormat ( "j son") . mode( "overwri t e") . save( "/t mp/my- j son- f i l
e. j son")

# i n Pyt hon csvFi l e. wri t e. f ormat ( "j son") . mode( "overwri t e") . save( "/t mp/my- j son- f i l
e. j son")

$ l s /t mp/my- j son- f i l e. j son/

/t mp/my- j son- f i l e. j son/part - 00000- t i d- 543. . . . j son
```

Parquet Files

Parquet is an open source column-oriented data store that provides a variety of storage optimizations, especially for analytics workloads. It provides columnar compression, which

saves storage space and allows for reading individual columns instead of entire files. It is a file format that works exceptionally well with Apache Spark and is in fact the default file format. We recommend writing data out to Parquet for long-term storage because reading from a Parquet file will always be more efficient than JSON or CSV. Another advantage of Parquet is that it supports complex types. This means that if your column is an array (which would fail with a CSV file, for example), map, or struct, you'll still be able to read and write that file without issue. Here's how to specify Parquet as the read format: `spark. read. f ormat ( "parquet ")`

Reading Parquet Files

Parquet has very few options because it enforces its own schema when storing data. Thus, all you need to set is the format and you are good to go. We can set the schema if we have strict requirements for what our DataFrame should look like. Oftentimes this is not necessary because we can use schema on read, which is similar to the i nf erSchema with CSV files. However, with Parquet files, this method is more powerful because the schema is built into the file itself (so no inference needed).

Here are some simple examples reading from parquet:

```
spark. read. f ormat ( "parquet ")

// i n Scal a spark. read. f ormat (
"parquet ")
  . l oad( "/dat a/f l i ght - dat a/parquet /2010- summary. parquet ") . show( 5)

# i n Pyt hon spark. read. f ormat (
"parquet ") \
  . l oad( "/dat a/f l i ght - dat a/parquet /2010- summary. parquet ") . show( 5)
```

Parquet options

As we just mentioned, there are very few Parquet options—precisely two, in fact—because it has a well-defined specification that aligns closely with the concepts in Spark. Table 9-5 presents the options.

> **WARNING**
>
> Even though there are only two options, you can still encounter problems if you're working with incompatible Parquet files. Be careful when you write out Parquet files with different versions of Spark (especially older ones) because this can cause significant headache.

Table 9-5. Parquet data source options

Read/Write Key	Potential	Default	Description

		Values		
Write	compression or codec	None, uncompressed, bzip2, deflate, gzip, lz4, or snappy	None	Declares what compression codec Spark should use to read or write the file.
Read	mergeSchema	true, false	Value of the configuration spark.sql.parquet.mergeSchema	You can incrementally add columns to newly written Parquet files in the same table/folder. Use this option to enable or disable this feature.

Writing Parquet Files

Writing Parquet is as easy as reading it. We simply specify the location for the file. The same partitioning rules apply:

```
// in Scala csvFile.write.format("parquet").mode("overwrite")  .save("/tmp/my-parquet-file.parquet")

# in Python csvFile.write.format("parquet").mode("overwrite") \  .save("/tmp/my-parquet-file.parquet")
```

ORC Files

ORC is a self-describing, type-aware columnar file format designed for Hadoop workloads. It is optimized for large streaming reads, but with integrated support for finding required rows quickly. ORC actually has no options for reading in data because Spark understands the file format quite well. An often-asked question is: What is the difference between ORC and Parquet? For the most part, they're quite similar; the fundamental difference is that Parquet is further optimized for use with Spark, whereas ORC is further optimized for Hive.

Reading Orc Files

Here's how to read an ORC file into Spark:

```
// in Scala spark.read.format("orc").load("/data/flight-data/orc/2010-summary.orc").show(5)

# in Python spark.read.format("orc").load("/data/flight-data/orc/2010-summary.orc").show(5)
```

Writing Orc Files

At this point in the chapter, you should feel pretty comfortable taking a guess at how to write ORC files. It really follows the exact same pattern that we have seen so far, in which we specify the format and then save the file:

```
// in Scala csvFile.write.format("orc").mode("overwrite").save("/tmp/my-json-file.orc")

# in Python
csvFile.write.format("orc").mode("overwrite").save("/tmp/my-json-file.orc")
```

SQL Databases

SQL datasources are one of the more powerful connectors because there are a variety of systems to which you can connect (as long as that system speaks SQL). For instance you can connect to a MySQL database, a PostgreSQL database, or an Oracle database. You also can connect to SQLite, which is what we'll do in this example. Of course, databases aren't just a set of raw files, so there are more options to consider regarding *how* you connect to the database. Namely you're going to need to begin considering things like authentication and connectivity (you'll need to determine whether the network of your Spark cluster is connected to the network of your database system).

To avoid the distraction of setting up a database for the purposes of this book, we provide a reference sample that runs on SQLite. We can skip a lot of these details by using SQLite, because it can work with minimal setup on your local machine with the limitation of not being able to work in a distributed setting. If you want to work through these examples in a distributed setting, you'll want to connect to another kind of database.

A PRIMER ON SQLITE

SQLite is the most used database engine in the entire world, and for good reason. It's powerful, fast, and easy to understand. This is because a SQLite database is just a file. That's going to make it very easy for you to get up and running because we include the source file in the official repository for this book. Simply download that file to your local machine, and you will be able to read from it and write to it. We're using SQLite, but all of the code here works with more traditional relational databases, as well, like MySQL. The primary difference is in the properties that you include when you connect to the database. When we're working with SQLite, there's no notion of user or password.

WARNING

Although SQLite makes for a good reference example, it's probablu not what you want to use in production. Also, SQLite will not necessarily work well in a distributed setting because of its requirement to lock the entire database on write. The example we present here will work in a similar way using MySQL or PostgreSQL, as well.

To read and write from these databases, you need to do two things: include the Java Database Connectivity (JDBC) driver for you particular database on the spark classpath, and provide the proper JAR for the driver itself. For example, to be able to read and write from PostgreSQL, you might run something like this:

```
. /bi n/spark- shel l \
- - dri ver- cl ass- pat h post gresql - 9. 4. 1207. j ar \
- - j ars post gresql - 9. 4. 1207. j ar
```

Just as with our other sources, there are a number of options that are available when reading from and writing to SQL databases. Only some of these are relevant for our current example, but Table 9-6 lists all of the options that you can set when working with JDBC databases.

Table 9-6. JDBC data source options

Property Name	Meaning
url	The JDBC URL to which to connect. The source-specific connection properties can be specified in the URL; for example, *jdbc:postgresql://localhost/test?user=fred&password=secret*.
dbt abl e	The JDBC table to read. Note that anything that is valid in a FROM clause of a SQL query can be used. For example, instead of a full table you could also use a subquery in parentheses.
dri ver	The class name of the JDBC driver to use to connect to this URL.
part i t i onCol umn, l owerBound, upperBound	If any one of these options is specified, then all others must be set as well. In addition, numPart i t i ons must be specified. These properties describe how to partition the table when reading in parallel from multiple workers. part i t i onCol umn must be a numeric column from the table in question. Notice that l owerBound and upperBound are used only to decide the partition stride, not for filtering the rows in the table. Thus, all rows in the table will be partitioned and returned. This option applies only to reading.
numPart i t i ons	The maximum number of partitions that can be used for parallelism in table reading and writing. This also determines the maximum number of concurrent JDBC connections. If the number of partitions to write exceeds this limit, we decrease it to this limit by calling coal esce(numPart i t i ons) before writing.
f et chsi ze	The JDBC fetch size, which determines how many rows to fetch per round trip. This can help performance on JDBC drivers, which default to low fetch size (e.g., Oracle with 10 rows). This option applies only to reading.

bat chsi ze	The JDBC batch size, which determines how many rows to insert per round trip. This can help performance on JDBC drivers. This option applies only to writing. The default is 1000.
i sol at i onLevel	The transaction isolation level, which applies to current connection. It can be one of NONE, READ_COMMI TTED, READ_UNCOMMI TTED, REPEATABLE_READ, or SERI ALI ZABLE, corresponding to standard transaction isolation levels defined by JDBC's Connect i on object. The default is READ_UNCOMMI TTED. This option applies only to writing. For more information, refer to the documentation in java.sql.Connection.
t runcat e	This is a JDBC writer-related option. When SaveMode. Overwri t e is enabled, Spark truncates an existing table instead of dropping and re-creating it. This can be more efficient, and it prevents the table metadata (e.g., indices) from being removed. However, it will not work in some cases, such as when the new data has a different schema. The default is f al se. This option applies only to writing.
creat eTabl eOpt i ons	This is a JDBC writer-related option. If specified, this option allows setting of database-specific table and partition options when creating a table (e.g., CREATE TABLE t (*name string*) ENGINE=InnoDB). This option applies only to writing.
creat eTabl eCol umnTypes	The database column data types to use instead of the defaults, when creating the table. Data type information should be specified in the same format as CREATE TABLE columns syntax (e.g., "name CHAR(64), comments VARCHAR(1024)"). The specified types should be valid Spark SQL data types. This option applies only to writing.

Reading from SQL Databases

When it comes to reading a file, SQL databases are no different from the other data sources that we looked at earlier. As with those sources, we specify the format and options, and then load in the data:

```
// i n Scal a val dri ver =  "org. sql i t e. J DBC" val pat h = "/dat
a/f l i ght - dat a/j dbc/my- sql i t e. db" val url = s"j dbc: sql i
t e: /${ pat h} " val t abl ename = "f l i ght _i nf o"

# i n Pyt hon dri ver = "org. sql i t e. J DBC" pat h = "/dat
a/f l i ght - dat a/j dbc/my- sql i t e. db" url = "j dbc: sql i
t e: " + pat h t abl ename = "f l i ght _i nf o"
```

After you have defined the connection properties, you can test your connection to the database itself to ensure that it is functional. This is an excellent troubleshooting technique to confirm that your database is available to (at the very least) the Spark driver. This is much less relevant for SQLite because that is a file on your machine but if you were using something like MySQL, you could test the connection with the following:

```
i mport j ava. sql . Dri verManager val connect i on = Dri
verManager. get Connect i on( url ) connect i on. i sCl osed( )
connect i on. cl ose( )
```

If this connection succeeds, you're good to go. Let's go ahead and read the DataFrame from the SQL table:

```
// in Scala val dbDataFrame = spark.read.format("jdbc").option("url",
url)
  .option("dbtable", tablename).option("driver", driver).load()

# in Python
dbDataFrame = spark.read.format("jdbc").option("url", url) \
  .option("dbtable", tablename).option("driver", driver).load()
```

SQLite has rather simple configurations (no users, for example). Other databases, like PostgreSQL, require more configuration parameters. Let's perform the same read that we just performed, except using PostgreSQL this time:

```
// in Scala val pgDF =
spark.read
  .format("jdbc")
  .option("driver", "org.postgresql.Driver")
  .option("url", "jdbc:postgresql://database_server")
  .option("dbtable", "schema.tablename")
  .option("user", "username").option("password", "my-secret-password").load()

# in Python pgDF = spark.read.format(
"jdbc") \
  .option("driver", "org.postgresql.Driver") \
  .option("url", "jdbc:postgresql://database_server") \
  .option("dbtable", "schema.tablename") \
  .option("user", "username").option("password", "my-secret-password").load()
```

As we create this DataFrame, it is no different from any other: you can query it, transform it, and join it without issue. You'll also notice that there is already a schema, as well. That's because Spark gathers this information from the table itself and maps the types to Spark data types. Let's get only the distinct locations to verify that we can query it as expected:

```
dbDataFrame.select("DEST_COUNTRY_NAME").distinct().show(5)

+--------------------+
|DEST_COUNTRY_NAME|
+--------------------+
|       Anguilla|
|        Russia|
|       Paraguay|
|        Senegal|
|        Sweden|
+--------------------+
```

Awesome, we can query the database! Before we proceed, there are a couple of nuanced details that are worth understanding.

Query Pushdown

First, Spark makes a best-effort attempt to filter data in the database itself before creating the DataFrame. For example, in the previous sample query, we can see from the query plan that it selects only the relevant column name from the table: `dbDat aFrame. sel ect ( "DEST_COUNTRY_NAME") . di st i nct ( ) . expl ai n`

```
== Physi cal Pl an ==
*HashAggregat e( keys=[ DEST_COUNTRY_NAME#8108] , f unct i ons=[ ] )
+- Exchange hashpart i t i oni ng( DEST_COUNTRY_NAME#8108, 200)
  +- *HashAggregat e( keys=[ DEST_COUNTRY_NAME#8108] , f unct i ons=[ ] )      +-
*Scan J DBCRel at i on( f l i ght _i nf o) [ numPart i t i ons=1] . . .
```

Spark can actually do better than this on certain queries. For example, if we specify a filter on our DataFrame, Spark will push that filter down into the database. We can see this in the explain plan under PushedFi l t ers.

```
// i n Scal a dbDat aFrame. f i l t er( "DEST_COUNTRY_NAME i n ( ' Angui l l a' , ' Sweden' ) ") .
expl ai n

# i n Pyt hon dbDat aFrame. f i l t er( "DEST_COUNTRY_NAME i n ( ' Angui l l a' , ' Sweden' ) ") .
expl ai n( )

== Physi cal Pl an ==
*Scan J DBCRel . . . PushedFi l t ers: [ *I n( DEST_COUNTRY_NAME, [ Angui l l a, Sweden] ) ] , . . .
```

Spark can't translate all of its own functions into the functions available in the SQL database in which you're working. Therefore, sometimes you're going to want to pass an entire query into your SQL that will return the results as a DataFrame. Now, this might seem like it's a bit complicated, but it's actually quite straightforward. Rather than specifying a table name, you just specify a SQL query. Of course, you do need to specify this in a special way; you must wrap the query in parenthesis and rename it to something—in this case, I just gave it the same table name:

```
// i n Scal a
val pushdownQuery = """( SELECT DI STI NCT( DEST_COUNTRY_NAME) FROM f l i ght _i nf o)  AS
f l i ght _i nf o""" val dbDat aFrame = spark. read. f ormat ( "j dbc")
  . opt i on( "url ", url ) . opt i on( "dbt abl e", pushdownQuery) . opt i on( "dri ver",  dri ver)   . l oad( )
# i n Pyt hon
pushdownQuery = """( SELECT DI STI NCT( DEST_COUNTRY_NAME) FROM f l i ght _i nf o)
AS f l i ght _i nf o""" dbDat aFrame = spark. read. f ormat ( "j dbc") \
```

```
  .option("url", url).option("dbtable", pushdownQuery).option("driver", driver) \
  .load()
```

Now when you query this table, you'll actually be querying the results of that query. We can see this in the explain plan. Spark doesn't even know about the actual schema of the table, just the one that results from our previous query: `dbDataFrame.explain()`

```
== Physical Plan ==
*Scan JDBCRelation(
( SELECT DISTINCT( DEST_COUNTRY_NAME)
  FROM flight_info) as flight_info
) [ numPartitions=1] [ DEST_COUNTRY_NAME#788] ReadSchema: . . .
```

Reading from databases in parallel

All throughout this book, we have talked about partitioning and its importance in data processing. Spark has an underlying algorithm that can read multiple files into one partition, or conversely, read multiple partitions out of one file, depending on the file size and the "splitability" of the file type and compression. The same flexibility that exists with files, also exists with SQL databases except that you must configure it a bit more manually. What you can configure, as seen in the previous options, is the ability to specify a maximum number of partitions to allow you to limit how much you are reading and writing in parallel:

```
// in Scala
val dbDataFrame = spark.read.format(
"jdbc")
  .option("url", url).option("dbtable", tablename).option("driver", driver)
  .option("numPartitions", 10).load()

# in Python
dbDataFrame = spark.read.format(
"jdbc") \
  .option("url", url).option("dbtable", tablename).option("driver", driver) \
  .option(
"numPartitions", 10).load()
```

In this case, this will still remain as one partition because there is not too much data. However, this configuration can help you ensure that you do not overwhelm the database when reading and writing data: `dbDataFrame.select("DEST_COUNTRY_NAME").distinct().show()`

There are several other optimizations that unfortunately only seem to be under another API set. You can explicitly push predicates down into SQL databases through the connection itself. This optimization allows you to control the physical location of certain data in certain partitions by specifying predicates. That's a mouthful, so let's look at a simple example. We only need data from two countries in our data: Anguilla and Sweden. We could filter these down and have them pushed into the database, but we can also go further by having them arrive in their own

partitions in Spark. We do that by specifying a list of predicates when we create the data source:

```
// i n Scal a
val props = new j ava. ut i l . Propert i es props. set Propert
y( "dri ver", "org. sql i t e. J DBC") val predi cat es = Array(
 "DEST_COUNTRY_NAME = ' Sweden' OR ORI GI N_COUNTRY_NAME = ' Sweden' ",
"DEST_COUNTRY_NAME = ' Angui l l a' OR ORI GI N_COUNTRY_NAME = ' Angui l l a' ") spark.
read. j dbc( url , t abl ename, predi cat es, props) . show( ) spark. read. j dbc( url , t abl ename,
predi cat es, props) . rdd. get NumPart i t i ons // 2

# i n Pyt hon props = { "dri ver": "org. sql i t e.
J DBC"} predi cat es = [
 "DEST_COUNTRY_NAME = ' Sweden' OR ORI GI N_COUNTRY_NAME = ' Sweden' ",
"DEST_COUNTRY_NAME = ' Angui l l a' OR ORI GI N_COUNTRY_NAME = ' Angui l l a' "] spark. read. j
dbc( url , t abl ename, predi cat es=predi cat es, propert i es=props) . show( ) spark. read. j dbc( url ,
t abl ename, predi cat es=predi cat es, propert i es=props) \   . rdd. get NumPart i t i ons( ) # 2

+- - - - - - - - - - - - - - - - +- - - - - - - - - - - - - - - - - - +- - - - - + |DEST_COUNTRY_NAME|ORI
GI N_COUNTRY_NAME|count |
+- - - - - - - - - - - - - - - - +- - - - - - - - - - - - - - - - - - +- - - - - +
|        Sweden|     Uni t ed St at es|   65|
|   Uni t ed St at es|          Sweden|   73|
|       Angui l l a|     Uni t ed St at es|   21|
|   Uni t ed St at es|         Angui l l a|   20|
+- - - - - - - - - - - - - - - - +- - - - - - - - - - - - - - - - - - +- - - - - +
```

If you specify predicates that are not disjoint, you can end up with lots of duplicate rows. Here's an example set of predicates that will result in duplicate rows:

```
// i n Scal a val props = new j ava. ut i l . Propert i es props.
set Propert y( "dri ver", "org. sql i t e. J DBC") val predi cat
es = Array(
 "DEST_COUNTRY_NAME ! = ' Sweden' OR ORI GI N_COUNTRY_NAME ! = ' Sweden' ",
"DEST_COUNTRY_NAME ! = ' Angui l l a' OR ORI GI N_COUNTRY_NAME ! = ' Angui l l a' ")
spark. read. j dbc( url , t abl ename, predi cat es, props) . count ( ) // 510

# i n Pyt hon props = { "dri ver": "org. sql i t e.
J DBC"} predi cat es = [
 "DEST_COUNTRY_NAME ! = ' Sweden' OR ORI GI N_COUNTRY_NAME ! = ' Sweden' ",
"DEST_COUNTRY_NAME ! = ' Angui l l a' OR ORI GI N_COUNTRY_NAME ! = ' Angui l l a' "] spark. read.
j dbc( url , t abl ename, predi cat es=predi cat es, propert i es=props) . count ( )
```

Partitioning based on a sliding window

Let's take a look to see how we can partition based on predicates. In this example, we'll partition based on our numerical count column. Here, we specify a minimum and a maximum for both the first partition and last partition. Anything outside of these bounds will be in the first partition or final partition. Then, we set the number of partitions we would like total (this is

the level of parallelism). Spark then queries our database in parallel and returns numPart i t i ons partitions. We simply modify the upper and lower bounds in order to place certain values in certain partitions. No filtering is taking place like we saw in the previous example:

```
// i n Scal a val col Name = "count " val l owerBound = 0L val upperBound = 348113L
// t hi s i s t he max count i n our dat abase val numPart i t i ons = 10

# i n Pyt hon col Name = "count " l owerBound = 0L upperBound = 348113L
# t hi s i s t he max count i n our dat abase numPart i t i ons = 10
```

This will distribute the intervals equally from low to high:

```
// i n Scal a spark. read. j dbc( url , t abl ename, col Name, l owerBound, upperBound, numPart i t i
ons, props)   . count ( ) // 255

# i n Pyt hon spark. read. j dbc( url , t abl ename, col umn=col Name, propert i
es=props,                    l owerBound=l owerBound, upperBound=upperBound,
numPart i t i ons=numPart i t i ons) . count ( ) # 255
```

Writing to SQL Databases

Writing out to SQL databases is just as easy as before. You simply specify the URI and write out the data according to the specified write mode that you want. In the following example, we specify overwrite, which overwrites the entire table. We'll use the CSV DataFrame that we defined earlier in order to do this:

```
// i n Scal a val newPat h = "j dbc: sql i t e: //t mp/my- sql i t e. db" csvFi l e. wri
t e. mode( "overwri t e") . j dbc( newPat h, t abl ename, props)

# i n Pyt hon newPat h = "j dbc: sql i t e: //t mp/my- sql i t e. db" csvFi l e. wri t e. j dbc( newPat
h, t abl ename, mode="overwri t e", propert i es=props)
```

Let's look at the results:

```
// i n Scal a spark. read. j dbc( newPat h, t abl ename, props) . count ( ) //
255

# i n Pyt hon
spark. read. j dbc( newPat h, t abl ename, propert i es=props) . count ( ) # 255
```

Of course, we can append to the table this new table just as easily:

```
// i n Scal a csvFi l e. wri t e. mode( "append") . j dbc( newPat h, t abl ename,
props)

# i n Pyt hon
```

```
csvFile.write.jdbc(newPath, tablename, mode="append", properties=props)
```
Notice that count increases:

```
// in Scala
spark.read.jdbc(newPath, tablename, props).count()
// 765

# in Python
spark.read.jdbc(newPath, tablename, properties=props).count() # 765
```

Text Files

Spark also allows you to read in plain-text files. Each line in the file becomes a record in the DataFrame. It is then up to you to transform it accordingly. As an example of how you would do this, suppose that you need to parse some Apache log files to some more structured format, or perhaps you want to parse some plain text for natural-language processing. Text files make a great argument for the Dataset API due to its ability to take advantage of the flexibility of native types.

Reading Text Files

Reading text files is straightforward: you simply specify the type to be textFile. With textFile, partitioned directory names are ignored. To read and write text files according to partitions, you should use text , which respects partitioning on reading and writing:

```
spark.read.textFile("/data/flight-data/csv/2010-summary.csv")
  .selectExpr("splitt(value, ',') as rows").show()

+--------------------+
|                rows|
+--------------------+
|[DEST_COUNTRY_NAM...|
|[United States, R...|
|[United States, A...|
|[Saint Vincent an...|
|[Italy, United St...|
+--------------------+
```

Writing Text Files

When you write a text file, you need to be sure to have only one string column; otherwise, the write will fail: `csvFile.select("DEST_COUNTRY_NAME").write.text("/tmp/simple-text-file.txt")`

If you perform some partitioning when performing your write (we'll discuss partitioning in the next couple of pages), you can write more columns. However, those columns will manifest as directories in the folder to which you're writing out to, instead of columns on every single file:

```
// in Scala csvFile.limit(10).select("DEST_COUNTRY_NAME",
"count")
  .write.partitionBy("count").text("/tmp/five-csv-files2.csv")

# in Python

csvFile.limit(10).select("DEST_COUNTRY_NAME", "count") \
  .write.partitionBy("count").text("/tmp/five-csv-files2py.csv")
```

Advanced I/O Concepts

We saw previously that we can control the parallelism of files that we write by controlling the partitions prior to writing. We can also control specific data layout by controlling two things: *bucketing* and *partitioning* (discussed momentarily).

Splittable File Types and Compression

Certain file formats are fundamentally "splittable." This can improve speed because it makes it possible for Spark to avoid reading an entire file, and access only the parts of the file necessary to satisfy your query. Additionally if you're using something like Hadoop Distributed File System (HDFS), splitting a file can provide further optimization if that file spans multiple blocks. In conjunction with this is a need to manage compression. Not all compression schemes are splittable. How you store your data is of immense consequence when it comes to making your Spark jobs run smoothly. We recommend Parquet with gzip compression.

Reading Data in Parallel

Multiple executors cannot read from the same file at the same time necessarily, but they can read different files at the same time. In general, this means that when you read from a folder with multiple files in it, each one of those files will become a partition in your DataFrame and be read in by available executors in parallel (with the remaining queueing up behind the others).

Writing Data in Parallel

The number of files or data written is dependent on the number of partitions the DataFrame has at the time you write out the data. By default, one file is written per partition of the data. This means that although we specify a "file," it's actually a number of files within a folder, with the name of the specified file, with one file per each partition that is written.

For example, the following code

```
csvFile.repartition(5).write.format("csv").save("/tmp/multiple.csv")
```

will end up with five files inside of that folder. As you can see from the list call:

```
ls /tmp/multiple.csv

/tmp/multiple.csv/part-00000-767df509-ec97-4740-8e15-4e173d365a8b.csv
/tmp/multiple.csv/part-00001-767df509-ec97-4740-8e15-4e173d365a8b.csv
/tmp/multiple.csv/part-00002-767df509-ec97-4740-8e15-4e173d365a8b.csv
/tmp/multiple.csv/part-00003-767df509-ec97-4740-8e15-4e173d365a8b.csv
/tmp/multiple.csv/part-00004-767df509-ec97-4740-8e15-4e173d365a8b.csv
```

Partitioning

Partitioning is a tool that allows you to control what data is stored (and where) as you write it. When you write a file to a partitioned directory (or table), you basically encode a column as a folder. What this allows you to do is skip lots of data when you go to read it in later, allowing you to read in only the data relevant to your problem instead of having to scan the complete dataset. These are supported for all file-based data sources:

```
// in Scala csvFile.limit(10).write.mode("overwrite").partitionBy(
"DEST_COUNTRY_NAME").save("/tmp/partitioned-files.parquet")

# in Python csvFile.limit(10).write.mode("overwrite").partitionBy(
"DEST_COUNTRY_NAME") \
 .save("/tmp/partitioned-files.parquet")
```

Upon writing, you get a list of folders in your Parquet "file":

```
$ ls /tmp/partitioned-files.parquet

...
DEST_COUNTRY_NAME=Costa Rica/
DEST_COUNTRY_NAME=Egypt/
DEST_COUNTRY_NAME=Equatorial Guinea/
DEST_COUNTRY_NAME=Senegal/
DEST_COUNTRY_NAME=United States/
```

Each of these will contain Parquet files that contain that data where the previous predicate was true:

```
$ ls /tmp/partitioned-files.parquet/DEST_COUNTRY_NAME=Senegal/part-

00000-tid.....parquet
```

This is probably the lowest-hanging optimization that you can use when you have a table that readers frequently filter by before manipulating. For instance, date is particularly common for a

partition because, downstream, often we want to look at only the previous week's data (instead of scanning the entire list of records). This can provide massive speedups for readers.

Bucketing

Bucketing is another file organization approach with which you can control the data that is specifically written to each file. This can help avoid shuffles later when you go to read the data because data with the same bucket ID will all be grouped together into one physical partition. This means that the data is prepartitioned according to how you expect to use that data later on, meaning you can avoid expensive shuffles when joining or aggregating.

Rather than partitioning on a specific column (which might write out a ton of directories), it's probably worthwhile to explore bucketing the data instead. This will create a certain number of files and organize our data into those "buckets":

```
val numberBuckets = 10 val col
umnToBucket By = "count "

csvFi l e. wri t e. f ormat ( "parquet ") . mode( "overwri t e")
  . bucket By( numberBuckets, col umnToBucket By) . saveAsTabl e( "bucket edFi l es")

$ l s /user/hi ve/warehouse/bucket edf i l es/

part - 00000- t i d- 1020575097626332666- 8. . . . parquet
part - 00000- t i d- 1020575097626332666- 8. . . . parquet
part - 00000- t i d- 1020575097626332666- 8. . . . parquet .
. .
```

Bucketing is supported only for Spark-managed tables. For more information on bucketing and partitioning, watch this talk from Spark Summit 2017.

Writing Complex Types

As we covered in Chapter 6, Spark has a variety of different internal types. Although Spark can work with all of these types, not every single type works well with every data file format. For instance, CSV files do not support complex types, whereas Parquet and ORC do.

Managing File Size

Managing file sizes is an important factor not so much for writing data but reading it later on. When you're writing lots of small files, there's a significant metadata overhead that you incur managing all of those files. Spark especially does not do well with small files, although many file systems (like HDFS) don't handle lots of small files well, either. You might hear this referred to as the "small file problem." The opposite is also true: you don't want files that are too large either, because it becomes inefficient to have to read entire blocks of data when you need only a few rows.

Spark 2.2 introduced a new method for controlling file sizes in a more automatic way. We saw previously that the number of output files is a derivative of the number of partitions we had at write time (and the partitioning columns we selected). Now, you can take advantage of another tool in order to limit output file sizes so that you can target an optimum file size. You can use the maxRecordsPerFi l e option and specify a number of your choosing. This allows you to better control file sizes by controlling the number of records that are written to each file. For example, if you set an option for a writer as df . wri t e. opt i on("maxRecordsPerFi l e", 5000) , Spark will ensure that files will contain at most 5,000 records.

Conclusion

In this chapter we discussed the variety of options available to you for reading and writing data in Spark. This covers nearly everything you'll need to know as an everyday user of Spark. For the curious, there are ways of implementing your own data source; however, we omitted instructions for how to do this because the API is currently evolving to better support Structured Streaming. If you're interested in seeing how to implement your own custom data sources, the Cassandra Connector is well organized and maintained and could provide a reference for the adventurous.

In Chapter 10, we discuss Spark SQL and how it interoperates with everything else we've seen so far in the Structured APIs.

Chapter 10. Spark SQL

Spark SQL is arguably one of the most important and powerful features in Spark. This chapter introduces the core concepts in Spark SQL that you need to understand. This chapter will not rewrite the ANSI-SQL specification or enumerate every single kind of SQL expression. If you read any other parts of this book, you will notice that we try to include SQL code wherever we include DataFrame code to make it easy to cross-reference with code samples. Other examples are available in the appendix and reference sections.

In a nutshell, with Spark SQL you can run SQL queries against views or tables organized into databases. You also can use system functions or define user functions and analyze query plans in order to optimize their workloads. This integrates directly into the DataFrame and Dataset API, and as we saw in previous chapters, you can choose to express some of your data manipulations in SQL and others in DataFrames and they will compile to the same underlying code.

What Is SQL?

SQL or *Structured Query Language* is a domain-specific language for expressing relational operations over data. It is used in all relational databases, and many "NoSQL" databases create their SQL dialect in order to make working with their databases easier. SQL is everywhere, and even though tech pundits prophesized its death, it is an extremely resilient data tool that many businesses depend on. Spark implements a subset of ANSI SQL:2003. This SQL standard is one that is available in the majority of SQL databases and this support means that Spark successfully runs the popular benchmark TPC-DS.

Big Data and SQL: Apache Hive

Before Spark's rise, Hive was the de facto big data SQL access layer. Originally developed at Facebook, Hive became an incredibly popular tool across industry for performing SQL operations on big data. In many ways it helped propel Hadoop into different industries because analysts could run SQL queries. Although Spark began as a general processing engine with Resilient Distributed Datasets (RDDs), a large cohort of users now use Spark SQL.

Big Data and SQL: Spark SQL

With the release of Spark 2.0, its authors created a superset of Hive's support, writing a native SQL parser that supports both ANSI-SQL as well as HiveQL queries. This, along with its unique interoperability with DataFrames, makes it a powerful tool for all sorts of companies. For example, in late 2016, Facebook announced that it had begun running Spark workloads and seeing large benefits in doing so. In the words of the blog post's authors:

> *We challenged Spark to replace a pipeline that decomposed to hundreds of Hive jobs into a single Spark job. Through a series of performance and reliability improvements, we were able to scale Spark to handle one of our entity ranking data processing use cases in production…. The Spark-based pipeline produced significant performance improvements (4.5–6x CPU, 3–4x resource reservation, and ~5x latency) compared with the old Hive-based pipeline, and it has been running in production for several months.*

The power of Spark SQL derives from several key facts: SQL analysts can now take advantage of Spark's computation abilities by plugging into the Thrift Server or Spark's SQL interface, whereas data engineers and scientists can use Spark SQL where appropriate in any data flow. This unifying API allows for data to be extracted with SQL, manipulated as a DataFrame, passed into one of Spark MLlibs' large-scale machine learning algorithms, written out to another data source, and everything in between.

NOTE

Spark SQL is intended to operate as an online analytic processing (OLAP) database, not an online transaction processing (OLTP) database. This means that it is not intended to perform extremely lowlatency queries. Even

though support for in-place modifications is sure to be something that comes up in the future, it's not something that is currently available.

Spark's Relationship to Hive

Spark SQL has a great relationship with Hive because it can connect to Hive metastores. The Hive metastore is the way in which Hive maintains table information for use across sessions. With Spark SQL, you can connect to your Hive metastore (if you already have one) and access table metadata to reduce file listing when accessing information. This is popular for users who are migrating from a legacy Hadoop environment and beginning to run all their workloads using Spark.

The Hive metastore

To connect to the Hive metastore, there are several properties that you'll need. First, you need to set the Metastore version (spark. sql . hi ve. met ast ore. versi on) to correspond to the proper Hive metastore that you're accessing. By default, this value is 1.2.1. You also need to set spark. sql . hi ve. met ast ore. j ars if you're going to change the way that the Hi veMet ast oreCl i ent is initialized. Spark uses the default versions, but you can also specify Maven repositories or a classpath in the standard format for the Java Virtual Machine (JVM). In addition, you might need to supply proper class prefixes in order to communicate with different databases that store the Hive metastore. You'll set these as shared prefixes that both Spark and Hive will share (spark. sql . hi ve. met ast ore. sharedPref i xes).

If you're connecting to your own metastore, it's worth checking the documentation for further updates and more information.

How to Run Spark SQL Queries

Spark provides several interfaces to execute SQL queries.

Spark SQL CLI

The Spark SQL CLI is a convenient tool with which you can make basic Spark SQL queries in local mode from the command line. Note that the Spark SQL CLI cannot communicate with the Thrift JDBC server. To start the Spark SQL CLI, run the following in the Spark directory:

```
. /bi n/spark- sql
```

You configure Hive by placing your *hive-site.xml*, *core-site.xml*, and *hdfs-site.xml* files in *conf/*. For a complete list of all available options, you can run . /bi n/spark- sql - - hel p.

Spark's Programmatic SQL Interface

In addition to setting up a server, you can also execute SQL in an ad hoc manner via any of

Spark's language APIs. You can do this via the method sql on the SparkSessi on object. This returns a DataFrame, as we will see later in this chapter. For example, in Python or Scala, we can run the following: spark. sql ("SELECT 1 + 1") . show( )

The command spark. sql ("SELECT 1 + 1") returns a DataFrame that we can then evaluate programmatically. Just like other transformations, this will not be executed eagerly but lazily. This is an immensely powerful interface because there are some transformations that are much simpler to express in SQL code than in DataFrames.

You can express multiline queries quite simply by passing a multiline string into the function. For example, you could execute something like the following code in Python or Scala:

```
spark. sql ( """SELECT user_i d, depart ment , f i rst _name FROM prof essors
  WHERE depart ment I N
    ( SELECT name FROM depart ment WHERE creat ed_dat e >= ' 2016- 01- 01' ) """)
```

Even more powerful, you can completely interoperate between SQL and DataFrames, as you see fit. For instance, you can create a DataFrame, manipulate it with SQL, and then manipulate it again as a DataFrame. It's a powerful abstraction that you will likely find yourself using quite a bit:

```
// i n Scal a spark. read. j son( "/dat a/f l i ght - dat a/j son/2015- summary.
j son")   . creat eOrRepl aceTempVi ew( "some_sql _vi ew") // DF => SQL
spark. sql ( """
SELECT DEST_COUNTRY_NAME, sum( count )
FROM some_sql _vi ew GROUP BY DEST_COUNTRY_NAME
""")
  . where( "DEST_COUNTRY_NAME l i ke ' S%' ") . where( "` sum( count ) ` > 10")
  . count ( ) // SQL => DF

# i n Pyt hon spark. read. j son( "/dat a/f l i ght - dat a/j son/2015- summary.
j son") \   . creat eOrRepl aceTempVi ew( "some_sql _vi ew") # DF => SQL

spark. sql ( """
SELECT DEST_COUNTRY_NAME, sum( count )
FROM some_sql _vi ew GROUP BY DEST_COUNTRY_NAME
""") \
  . where( "DEST_COUNTRY_NAME l i ke ' S%' ") . where( "` sum( count ) ` > 10") \   .
count ( ) # SQL => DF
```

SparkSQL Thrift JDBC/ODBC Server

Spark provides a Java Database Connectivity (JDBC) interface by which either you or a remote program connects to the Spark driver in order to execute Spark SQL queries. A common use

case might be a for a business analyst to connect business intelligence software like Tableau to Spark. The Thrift JDBC/Open Database Connectivity (ODBC) server implemented here corresponds to the HiveServer2 in Hive 1.2.1. You can test the JDBC server with the beeline script that comes with either Spark or Hive 1.2.1.

To start the JDBC/ODBC server, run the following in the Spark directory:

```
. /sbi n/st art - t hri f t server. sh
```

This script accepts all bi n/spark- submi t command-line options. To see all available options for configuring this Thrift Server, run . /sbi n/st art - t hri f t server. sh - - hel p. By default, the server listens on localhost:10000. You can override this through environmental variables or system properties.

For environment configuration, use this:

```
export HI VE_SERVER2_THRI FT_PORT=<l i st eni ng- port > export
HI VE_SERVER2_THRI FT_BI ND_HOST=<l i st eni ng- host >
. /sbi n/st art - t hri f t server. sh \   - -
mast er <mast er- uri > \   . . .
```

For system properties:

```
. /sbi n/st art - t hri f t server. sh \
- - hi veconf hi ve. server2. t hri f t . port =<l i st eni ng- port > \
- - hi veconf hi ve. server2. t hri f t . bi nd. host =<l i st eni ng- host > \
- - mast er <mast er- uri >
  . . .
```

You can then test this connection by running the following commands:

```
. /bi n/beel i ne

beel i ne> ! connect j dbc: hi ve2: //l ocal host : 10000
```

Beeline will ask you for a username and password. In nonsecure mode, simply type the username on your machine and a blank password. For secure mode, follow the instructions given in the beeline documentation.

Catalog

The highest level abstraction in Spark SQL is the Catalog. The Catalog is an abstraction for the storage of metadata about the data stored in your tables as well as other helpful things like databases, tables, functions, and views. The catalog is available in the org. apache. spark. sql . cat al og. Cat al og package and contains a number of helpful functions for doing things like

listing tables, databases, and functions. We will talk about all of these things shortly. It's very self-explanatory to users, so we will omit the code samples here but it's really just another programmatic interface to Spark SQL. This chapter shows only the SQL being executed; thus, if you're using the programmatic interface, keep in mind that you need to wrap everything in a spark. sql function call to execute the relevant code.

Tables

To do anything useful with Spark SQL, you first need to define tables. Tables are logically equivalent to a DataFrame in that they are a structure of data against which you run commands. We can join tables, filter them, aggregate them, and perform different manipulations that we saw in previous chapters. The core difference between tables and DataFrames is this: you define DataFrames in the scope of a programming language, whereas you define tables within a database. This means that when you create a table (assuming you never changed the database), it will belong to the *default* database. We discuss databases more fully later on in the chapter.

An important thing to note is that in Spark 2.X, tables *always contain data*. There is no notion of a temporary table, only a view, which does not contain data. This is important because if you go to drop a table, you can risk losing the data when doing so.

Spark-Managed Tables

One important note is the concept of *managed* versus *unmanaged* tables. Tables store two important pieces of information. The data within the tables as well as the data about the tables; that is, the *metadata*. You can have Spark manage the metadata for a set of files as well as for the data. When you define a table from files on disk, you are defining an unmanaged table. When you use saveAsTabl e on a DataFrame, you are creating a managed table for which Spark will track of all of the relevant information.

This will read your table and write it out to a new location in Spark format. You can see this reflected in the new explain plan. In the explain plan, you will also notice that this writes to the default Hive warehouse location. You can set this by setting the spark. sql . warehouse. di r configuration to the directory of your choosing when you create your SparkSession. By default Spark sets this to /user/hi ve/warehouse:

Note in the results that a database is listed. Spark also has databases which we will discuss later in this chapter, but for now you should keep in mind that you can also see tables in a specific database by using the query *show t abl es I N dat abaseName*, where *dat abaseName* represents the name of the database that you want to query.

If you are running on a new cluster or local mode, this should return zero results.

Creating Tables

You can create tables from a variety of sources. Something fairly unique to Spark is the capability of reusing the entire Data Source API within SQL. This means that you do not need to define a table and then load data into it; Spark lets you create one on the fly. You can even specify all sorts of sophisticated options when you read in a file. For example, here's a simple way to read in the flight data we worked with in previous chapters:

```
CREATE TABLE f l i ght s (
  DEST_COUNTRY_NAME STRI NG, ORI GI N_COUNTRY_NAME STRI NG, count LONG)
USI NG J SON OPTI ONS ( pat h ' /dat a/f l i ght - dat a/j son/2015- summary. j son' )
```

USING AND STORED AS

The specification of the USI NG syntax in the previous example is of significant importance. If you do not specify the format, Spark will default to a Hive SerDe configuration. This has performance implications for future readers and writers because Hive SerDes are much slower than Spark's native serialization. Hive users can also use the STORED AS syntax to specify that this should be a Hive table.

You can also add comments to certain columns in a table, which can help other developers understand the data in the tables:

```
CREATE TABLE f l i ght s_csv (
  DEST_COUNTRY_NAME STRI NG,
  ORI GI N_COUNTRY_NAME STRI NG COMMENT "remember, t he US wi l l be most preval ent ",   count
LONG)
USI NG csv OPTI ONS ( header t rue, pat h ' /dat a/f l i ght - dat a/csv/2015- summary. csv' )
```

It is possible to create a table from a query as well:

```
CREATE TABLE f l i ght s_f rom_sel ect USI NG parquet AS SELECT * FROM f l i ght s
```

In addition, you can specify to create a table only if it does not currently exist:

NOTE

In this example, we are creating a Hive-compatible table because we did not explicitly specify the format via USI NG. We can also do the following:

```
CREATE TABLE I F NOT EXI STS f l i ght s_f rom_sel ect
  AS SELECT * FROM f l i ght s
```

Finally, you can control the layout of the data by writing out a partitioned dataset, as we saw in Chapter 9:

```
CREATE TABLE part i t i oned_f l i ght s USI NG parquet PARTI TI ONED BY ( DEST_COUNTRY_NAME)
AS SELECT DEST_COUNTRY_NAME, ORI GI N_COUNTRY_NAME, count FROM f l i ght s LI MI T 5
```

These tables will be available in Spark even through sessions; temporary tables do not currently exist in Spark. You must create a temporary view, which we demonstrate later in this chapter.

Creating External Tables

As we mentioned in the beginning of this chapter, Hive was one of the first big data SQL systems, and Spark SQL is completely compatible with Hive SQL (HiveQL) statements. One of the use cases that you might encounter is to port your legacy Hive statements to Spark SQL. Luckily, you can, for the most part, just copy and paste your Hive statements directly into Spark SQL. For example, in the example that follows, we create an *unmanaged table*. Spark will manage the table's metadata; however, the files are not managed by Spark at all. You create this table by using the CREATE EXTERNAL TABLE statement.

You can view any files that have already been defined by running the following command:

```
CREATE EXTERNAL TABLE hi ve_f l i ght s (
 DEST_COUNTRY_NAME STRI NG, ORI GI N_COUNTRY_NAME STRI NG, count LONG)
ROW FORMAT DELI MI TED FI ELDS TERMI NATED BY ' , ' LOCATI ON ' /dat a/f l i ght - dat a- hi ve/' You
```

can also create an external table from a select clause:

```
CREATE EXTERNAL TABLE hi ve_f l i ght s_2
ROW FORMAT DELI MI TED FI ELDS TERMI NATED BY ' , '
LOCATI ON ' /dat a/f l i ght - dat a- hi ve/' AS SELECT * FROM f l i ght s
```

Inserting into Tables

Insertions follow the standard SQL syntax:

```
I NSERT I NTO f l i ght s_f rom_sel ect
 SELECT DEST_COUNTRY_NAME, ORI GI N_COUNTRY_NAME, count FROM f l i ght s LI MI T 20
```

You can optionally provide a partition specification if you want to write only into a certain partition. Note that a write will respect a partitioning scheme, as well (which may cause the above query to run quite slowly); however, it will add additional files only into the end partitions:

```
I NSERT I NTO part i t i oned_f l i ght s
 PARTI TI ON ( DEST_COUNTRY_NAME="UNI TED STATES")
 SELECT count , ORI GI N_COUNTRY_NAME FROM f l i ght s
 WHERE DEST_COUNTRY_NAME=' UNI TED STATES' LI MI T 12
```

Describing Table Metadata

We saw earlier that you can add a comment when creating a table. You can view this by describing the table metadata, which will show us the relevant comment:

```
DESCRI BE TABLE f l i ght s_csv
```

You can also see the partitioning scheme for the data by using the following (note, however, that this works only on partitioned tables):

```
SHOW PARTI TI ONS part i t i oned_f l i ght s
```

Refreshing Table Metadata

Maintaining table metadata is an important task to ensure that you're reading from the most recent set of data. There are two commands to refresh table metadata. REFRESH TABLE refreshes all cached entries (essentially, files) associated with the table. If the table were previously cached, it would be cached lazily the next time it is scanned:

```
REFRESH t abl e part i t i oned_f l i ght s
```

Another related command is REPAI R TABLE, which refreshes the partitions maintained in the catalog for that given table. This command's focus is on collecting new partition information—an example might be writing out a new partition manually and the need to repair the table accordingly:

```
MSCK REPAI R TABLE part i t i oned_f l i ght s
```

Dropping Tables

You cannot delete tables: you can only "drop" them. You can drop a table by using the DROP keyword. If you drop a managed table (e.g., f l i ght s_csv), both the data and the table definition will be removed:

```
DROP TABLE f l i ght s_csv;
```

WARNING

Dropping a table deletes the data in the table, so you need to be very careful when doing this.

If you try to drop a table that does not exist, you will receive an error. To only delete a table if it already exists, use DROP TABLE I F EXI STS.

```
DROP TABLE I F EXI STS f l i ght s_csv;
```

WARNING

This deletes the data in the table, so exercise caution when doing this.

Dropping unmanaged tables

If you are dropping an unmanaged table (e.g., hi ve_f l i ght s), no data will be removed but you will no longer be able to refer to this data by the table name.

Caching Tables

Just like DataFrames, you can cache and uncache tables. You simply specify which table you would like using the following syntax:

```
CACHE TABLE f l i ght s
```

Here's how you uncache them:

```
UNCACHE TABLE FLI GHTS
```

Views

Now that you created a table, another thing that you can define is a view. A view specifies a set of transformations on top of an existing table—basically just saved query plans, which can be convenient for organizing or reusing your query logic. Spark has several different notions of views. Views can be global, set to a database, or per session.

Creating Views

To an end user, views are displayed as tables, except rather than rewriting all of the data to a new location, they simply perform a transformation on the source data at query time. This might be a f i l t er, sel ect , or potentially an even larger GROUP BY or ROLLUP. For instance, in the following example, we create a view in which the destination is Uni t ed St at es in order to see only those flights:

```
CREATE VI EW j ust _usa_vi ew AS
  SELECT * FROM f l i ght s WHERE dest _count ry_name = ' Uni t ed St at es'
```

Like tables, you can create temporary views that are available only during the current session and are not registered to a database:

```
CREATE TEMP VI EW j ust _usa_vi ew_t emp AS
  SELECT * FROM f l i ght s WHERE dest _count ry_name = ' Uni t ed St at es'
```

Or, it can be a global temp view. Global temp views are resolved regardless of database and are viewable across the entire Spark application, but they are removed at the end of the session:

```
CREATE GLOBAL TEMP VI EW j ust _usa_gl obal _vi ew_t emp AS
 SELECT * FROM f l i ght s WHERE dest _count ry_name = ' Uni t ed St at es'

SHOW TABLES
```

You can also specify that you would like to overwite a view if one already exists by using the keywords shown in the sample that follows. We can overwrite both temp views and regular views:

```
CREATE OR REPLACE TEMP VI EW j ust _usa_vi ew_t emp AS
  SELECT * FROM f l i ght s WHERE dest _count ry_name = ' Uni t ed St at es' Now you
```

can query this view just as if it were another table:

```
SELECT * FROM j ust _usa_vi ew_t emp
```

A view is effectively a transformation and Spark will perform it only at query time. This means that it will only apply that filter after you actually go to query the table (and not earlier). Effectively, views are equivalent to creating a new DataFrame from an existing DataFrame.

In fact, you can see this by comparing the query plans generated by Spark DataFrames and Spark

SQL. In DataFrames, we would write the following: val f l i ght s = spark. read. f ormat ("j son")

```
 . l oad( "/dat a/f l i ght - dat a/j son/2015- summary. j son") val j ust _usa_df = f l i ght s.
where( "dest _count ry_name = ' Uni t ed St at es' ") j ust _usa_df . sel ect Expr( "*") .
expl ai n
```

In SQL, we would write (querying from our view) this:

```
EXPLAI N SELECT * FROM j ust _usa_vi ew Or,
```

equivalently:

```
EXPLAI N SELECT * FROM f l i ght s WHERE dest _count ry_name = ' Uni t ed St at es'
```

Due to this fact, you should feel comfortable in writing your logic either on DataFrames or SQL —whichever is most comfortable and maintainable for you.

Dropping Views

You can drop views in the same way that you drop tables; you simply specify that what you intend to drop is a *view* instead of a table. The main difference between dropping a view and

dropping a table is that with a view, no underlying data is removed, only the view definition itself:

```
DROP VI EW I F EXI STS j ust _usa_vi ew;
```

Databases

Databases are a tool for organizing tables. As mentioned earlier, if you do not define one, Spark will use the default database. Any SQL statements that you run from within Spark (including DataFrame commands) execute within the context of a database. This means that if you change the database, any user-defined tables will remain in the previous database and will need to be queried differently.

> **WARNING**
>
> This can be a source of confusion, especially if you're sharing the same context or session for your coworkers, so be sure to set your databases appropriately.

You can see all databases by using the following command:

```
SHOW DATABASES
```

Creating Databases

Creating databases follows the same patterns you've seen previously in this chapter; however, here you use the CREATE DATABASE keywords:

```
CREATE DATABASE some_db
```

Setting the Database

You might want to set a database to perform a certain query. To do this, use the USE keyword followed by the database name:

```
USE some_db
```

After you set this database, all queries will try to resolve table names to this database. Queries that were working just fine might now fail or yield different results because you are in a different database:

```
SHOW t abl es

SELECT * FROM f l i ght s - - f ai l s wi t h t abl e/vi ew not f ound
```

However, you can query different databases by using the correct prefix:

```
SELECT * FROM default.flights
```

You can see what database you're currently using by running the following command:

```
SELECT current_database()
```

You can, of course, switch back to the default database:

```
USE default;
```

Dropping Databases

Dropping or removing databases is equally as easy: you simply use the DROP DATABASE keyword:

```
DROP DATABASE IF EXISTS some_db;
```

Select Statements

Queries in Spark support the following ANSI SQL requirements (here we list the layout of the SELECT expression):

```
SELECT [ALL|DISTINCT] named_expression[, named_expression, ...]
    FROM relation[, relation, ...]
    [lateral_view[, lateral_view, ...]]
    [WHERE boolean_expression]
    [aggregation [HAVING boolean_expression]]
    [ORDER BY sort_expressions]
    [CLUSTER BY expressions]
    [DISTRIBUTE BY expressions]
    [SORT BY sort_expressions]
    [WINDOW named_window[, WINDOW named_window, ...]]
    [LIMIT num_rows]

named_expression:
    : expression [AS alias]

relation:
    | join_relation
    | (table_name|query|relation) [sample] [AS alias]
    : VALUES (expressions) [, (expressions), ...]
        [AS (column_name[, column_name, ...])]

expressions:
    : expression[, expression, ...]
```

```
sort _expressi ons:
   : expressi on [ ASC|DESC] [ , expressi on [ ASC|DESC] , . . . ]
```

case…when…then Statements

Oftentimes, you might need to conditionally replace values in your SQL queries. You can do this by using a case. . . when. . . t hen. . . end style statement. This is essentially the equivalent of programmatic i f statements:

```
SELECT
 CASE WHEN DEST_COUNTRY_NAME = ' UNI TED STATES' THEN 1
    WHEN DEST_COUNTRY_NAME = ' Egypt ' THEN 0
    ELSE - 1 END FROM part i t
i oned_f l i ght s
```

Advanced Topics

Now that we defined where data lives and how to organize it, let's move on to querying it. A SQL query is a SQL statement requesting that some set of commands be run. SQL statements can define manipulations, definitions, or controls. The most common case are the manipulations, which is the focus of this book.

Complex Types

Complex types are a departure from standard SQL and are an incredibly powerful feature that does not exist in standard SQL. Understanding how to manipulate them appropriately in SQL is essential. There are three core complex types in Spark SQL: structs, lists, and maps.

Structs

Structs are more akin to maps. They provide a way of creating or querying nested data in Spark. To create one, you simply need to wrap a set of columns (or expressions) in parentheses:

```
CREATE VI EW I F NOT EXI STS nest ed_dat a AS
 SELECT ( DEST_COUNTRY_NAME, ORI GI N_COUNTRY_NAME) as count ry, count FROM f l i ght s
```

Now, you can query this data to see what it looks like:

```
SELECT * FROM nest ed_dat a
```

You can even query individual columns within a struct—all you need to do is use dot syntax:

```
SELECT count ry. DEST_COUNTRY_NAME, count FROM nest ed_dat a
```

If you like, you can also select all the subvalues from a struct by using the struct's name and select all of the subcolumns. Although these aren't truly subcolumns, it does provide a simpler way to think about them because we can do everything that we like with them as if they were a column:

```
SELECT count ry. *, count FROM nest ed_dat a
```

Lists

If you're familiar with lists in programming languages, Spark SQL lists will feel familiar. There are several ways to create an array or list of values. You can use the col l ect _l i st function, which creates a list of values. You can also use the function col l ect _set , which creates an array without duplicate values. These are both aggregation functions and therefore can be specified only in aggregations:

```
SELECT DEST_COUNTRY_NAME as new_name, col l ect _l i st ( count ) as f l i ght _count s,   col
l ect _set ( ORI GI N_COUNTRY_NAME) as ori gi n_set FROM f l i ght s GROUP BY
DEST_COUNTRY_NAME
```

You can, however, also create an array manually within a column, as shown here:

```
SELECT DEST_COUNTRY_NAME, ARRAY( 1, 2, 3) FROM f l i ght s
```

You can also query lists by position by using a Python-like array query syntax:

```
SELECT DEST_COUNTRY_NAME as new_name, col l ect _l i st ( count ) [ 0]
FROM f l i ght s GROUP BY DEST_COUNTRY_NAME
```

You can also do things like convert an array back into rows. You do this by using the expl ode function. To demonstrate, let's create a new view as our aggregation:

```
CREATE OR REPLACE TEMP VI EW f l i ght s_agg AS
 SELECT DEST_COUNTRY_NAME, col l ect _l i st ( count ) as col l ect ed_count s   FROM f l i
ght s GROUP BY DEST_COUNTRY_NAME
```

Now let's explode the complex type to one row in our result for every value in the array. The DEST_COUNTRY_NAME will duplicate for every value in the array, performing the exact opposite of the original col l ect and returning us to the original DataFrame:

```
SELECT expl ode( col l ect ed_count s) , DEST_COUNTRY_NAME FROM f l i ght s_agg
```

Functions

In addition to complex types, Spark SQL provides a variety of sophisticated functions. You can find most of these functions in the DataFrames function reference; however, it is worth

understanding how to find these functions in SQL, as well. To see a list of functions in Spark SQL, you use the SHOW FUNCTI ONS statement:

```
SHOW FUNCTI ONS
```

You can also more specifically indicate whether you would like to see the system functions (i.e., those built into Spark) as well as user functions:

```
SHOW SYSTEM FUNCTI ONS
```

User functions are those defined by you or someone else sharing your Spark environment. These are the same user-defined functions that we talked about in earlier chapters (we will discuss how to create them later on in this chapter):

```
SHOW USER FUNCTI ONS
```

You can filter all SHOW commands by passing a string with wildcard (*) characters. Here, we can see all functions that begin with “s”:

```
SHOW FUNCTI ONS "s*";
```

Optionally, you can include the LI KE keyword, although this is not necessary:

```
SHOW FUNCTI ONS LI KE "col l ect *";
```

Even though listing functions is certainly useful, often you might want to know more about specific functions themselves. To do this, use the DESCRI BE keyword, which returns the documentation for a specific function.

User-defined functions

As we saw in Chapters 3 and 4, Spark gives you the ability to define your own functions and use them in a distributed manner. You can define functions, just as you did before, writing the function in the language of your choice and then registering it appropriately:

```
def power3( number: Doubl e) : Doubl e = number * number * number spark. udf
. regi st er( "power3", power3( _: Doubl e) : Doubl e)

SELECT count , power3( count ) FROM f l i ght s
```

You can also register functions through the Hive CREATE TEMPORARY FUNCTI ON syntax.

Subqueries

With subqueries, you can specify queries within other queries. This makes it possible for you to specify some sophisticated logic within your SQL. In Spark, there are two fundamental subqueries. *Correlated subqueries* use some information from the outer scope of the query in order to supplement information in the subquery. *Uncorrelated subqueries* include no information from the outer scope. Each of these queries can return one (scalar subquery) or more values. Spark also includes support for *predicate subqueries*, which allow for filtering based on values.

Uncorrelated predicate subqueries

For example, let's take a look at a predicate subquery. In this example, this is composed of two *uncorrelated* queries. The first query is just to get the top five country destinations based on the data we have:

```
SELECT dest _count ry_name FROM f l i ght s
GROUP BY dest _count ry_name ORDER BY sum( count ) DESC LI MI T 5 This
```

gives us the following result:

```
+- - - - - - - - - - - - - - - - - + |dest
_count ry_name|
+- - - - - - - - - - - - - - - - - +
|   Uni t ed St at es|
|        Canada|
|        Mexi co|
|  Uni t ed Ki ngdom|
|        J apan|
+- - - - - - - - - - - - - - - - - +
```

Now we place this subquery inside of the filter and check to see if our origin country exists in that list:

```
SELECT * FROM f l i ght s
WHERE ori gi n_count ry_name I N ( SELECT dest _count ry_name FROM f l i ght s
    GROUP BY dest _count ry_name ORDER BY sum( count ) DESC LI MI T 5)
```

This query is uncorrelated because it does not include any information from the outer scope of the query. It's a query that you can run on its own.

Correlated predicate subqueries

Correlated predicate subqueries allow you to use information from the outer scope in your inner query. For example, if you want to see whether you have a flight that will take you back from your destination country, you could do so by checking whether there is a flight that has the destination country as an origin and a flight that had the origin country as a destination:

```
SELECT * FROM f l i ght s f 1
WHERE EXI STS ( SELECT 1 FROM f l i ght s f 2
        WHERE f 1. dest _count ry_name = f 2. ori gi n_count ry_name) AND EXI STS (
SELECT 1 FROM f l i ght s f 2
        WHERE f 2. dest _count ry_name = f 1. ori gi n_count ry_name)
```

EXI STS just checks for some existence in the subquery and returns true if there is a value. You can flip this by placing the NOT operator in front of it. This would be equivalent to finding a flight to a destination from which you won't be able to return!

Uncorrelated scalar queries

Using uncorrelated scalar queries, you can bring in some supplemental information that you might not have previously. For example, if you wanted to include the maximum value as its own column from the entire counts dataset, you could do this:

```
SELECT *, ( SELECT max( count ) FROM f l i ght s) AS maxi mum FROM f l i ght s
```

Miscellaneous Features

There are some features in Spark SQL that don't quite fit in previous sections of this chapter, so we're going to include them here in no particular order. These can be relevant when performing optimizations or debugging your SQL code.

Configurations

There are several Spark SQL application configurations, which we list in Table 10-1. You can set these either at application initialization or over the course of application execution (like we have seen with shuffle partitions throughout this book).

Table 10-1. Spark SQL configurations

Table 10-1. Spark SQL configurations

Property Name	Default	Meaning
spark. sql . i nMemoryCol umnarSt orage. compressed	t rue	When set to true, Spark SQL automatically selects a compression codec for each column based on statistics of the data.
spark. sql . i nMemoryCol umnarSt orage. bat chSi ze	10000	Controls the size of batches for columnar caching. Larger batch sizes can improve memory utilization and compression, but risk Out Of MemoryErrors (OOMs) when caching data.
spark. sql . f i l es. maxPart i t i onByt es	134217728 (128 MB)	The maximum number of bytes to pack into a single partition when reading files.

spark. sql . f i l es. openCost I nByt es	4194304 (4 MB)	The estimated cost to open a file, measured by the number of bytes that could be scanned in the same time. This is used when putting multiple files into a partition. It is better to overestimate; that way the partitions with small files will be faster than partitions with bigger files (which is scheduled first).
spark. sql . broadcast Ti meout	300	Timeout in seconds for the broadcast wait time in broadcast joins.
spark. sql . aut oBroadcast J oi nThreshol d	10485760 (10 MB)	Configures the maximum size in bytes for a table that will be broadcast to all worker nodes when performing a join. You can disable broadcasting by setting this value to -1. Note that currently statistics are supported only for Hive Metastore tables for which the command ANALYZE TABLE COMPUTE STATI STI CS noscan has been run.
spark. sql . shuf f l e. part i t i ons	200	Configures the number of partitions to use when shuffling data for joins or aggregations.

Setting Configuration Values in SQL

We talk about configurations in Chapter 15, but as a preview, it's worth mentioning how to set configurations from SQL. Naturally, you can only set Spark SQL configurations that way, but here's how you can set shuffle partitions:

```
SET spark. sql . shuf f l e. part i t i ons=20
```

Conclusion

It should be clear from this chapter that Spark SQL and DataFrames are very closely related and that you should be able to use nearly all of the examples throughout this book with only small syntactical tweaks. This chapter illustrated more of the Spark SQL–related specifics. Chapter 11 focuses on a new concept: Datasets that allow for type-safe structured transformations.

Chapter 11. Datasets

Datasets are the foundational type of the Structured APIs. We already worked with DataFrames, which are Datasets of type Row, and are available across Spark's different languages. Datasets are a strictly Java Virtual Machine (JVM) language feature that work only with Scala and Java. Using Datasets, you can define the object that each row in your Dataset will consist of. In Scala, this will be a case class object that essentially defines a schema that you can use, and in Java, you will define a Java Bean. Experienced users often refer to Datasets as the "typed set of APIs" in Spark. For more information, see Chapter 4.

In Chapter 4, we discussed that Spark has types like St ri ngType, Bi gI nt Type, St ruct Type, and so on. Those Spark-specific types map to types available in each of Spark's languages like St ri ng, I nt eger, and Doubl e. When you use the DataFrame API, you do not create strings or integers, but Spark manipulates the data for you by manipulating the Row object. In fact, if you use Scala or Java, all "DataFrames" are actually Datasets of type Row. To efficiently support domain-specific objects, a special concept called an "Encoder" is required. The encoder maps the domain-specific type T to Spark's internal type system.

For example, given a class Person with two fields, name (string) and age (int), an encoder directs Spark to generate code at runtime to serialize the Person object into a binary structure. When using DataFrames or the "standard" Structured APIs, this binary structure will be a Row. When we want to create our own domain-specific objects, we specify a case cl ass in Scala or a J avaBean in Java. Spark will allow us to manipulate this object (in place of a Row) in a distributed manner.

When you use the Dataset API, for every row it touches, this domain specifies type, Spark converts the Spark Row format to the object you specified (a case class or Java class). This conversion slows down your operations but can provide more flexibility. You will notice a hit in performance but this is a far different order of magnitude from what you might see from something like a user-defined function (UDF) in Python, because the performance costs are not as extreme as switching programming languages, but it is an important thing to keep in mind.

When to Use Datasets

You might ponder, if I am going to pay a performance penalty when I use Datasets, why should I use them at all? If we had to condense this down into a canonical list, here are a couple of reasons:

- When the operation(s) you would like to perform cannot be expressed using DataFrame manipulations

- When you want or need type-safety, and you're willing to accept the cost of performance to achieve it

Let's explore these in more detail. There are some operations that cannot be expressed using the Structured APIs we have seen in the previous chapters. Although these are not particularly common, you might have a large set of business logic that you'd like to encode in one specific function instead of in SQL or DataFrames. This is an appropriate use for Datasets. Additionally, the Dataset API is type-safe. Operations that are not valid for their types, say subtracting two string types, will fail at compilation time not at runtime. If correctness and bulletproof code is your highest priority, at the cost of some performance, this can be a great choice for you. This does not protect you from malformed data but can allow you to more elegantly handle and organize it.

Another potential time for which you might want to use Datasets is when you would like to reuse a variety of transformations of entire rows between single-node workloads and Spark workloads. If you have some experience with Scala, you might notice that Spark's APIs reflect those of Scala Sequence Types, but they operate in a distributed fashion. In fact, Martin Odersky, the inventor of Scala, said just that in 2015 at Spark Summit Europe. Due to this, one advantage of using Datasets is that if you define all of your data and transformations as accepting case classes it is trivial to reuse them for both distributed and local workloads. Additionally, when you collect your DataFrames to local disk, they will be of the correct class and type, sometimes making further manipulation easier.

Probably the most popular use case is to use DataFrames and Datasets in tandem, manually trading off between performance and type safety when it is most relevant for your workload. This might be at the end of a large, DataFrame-based extract, transform, and load (ETL) transformation when you'd like to collect data to the driver and manipulate it by using singlenode libraries, or it might be at the beginning of a transformation when you need to perform perrow parsing before performing filtering and further manipulation in Spark SQL.

Creating Datasets

Creating Datasets is somewhat of a manual operation, requiring you to know and define the schemas ahead of time.

In Java: Encoders

Java Encoders are fairly simple, you simply specify your class and then you'll encode it when you come upon your DataFrame (which is of type Dat aset <Row>):

```
i mport org. apache. spark. sql . Encoders;

publ i c cl ass Fl i ght i mpl ement s Seri al i zabl e{
  St ri ng DEST_COUNTRY_NAME;
```

```
  St ri ng ORI GI N_COUNTRY_NAME;
  Long DEST_COUNTRY_NAME;
}
Dat aset <Fl i ght > f l i ght s = spark. read
  . parquet ( "/dat a/f l i ght - dat a/parquet /2010- summary. parquet /")
  . as( Encoders. bean( Fl i ght . cl ass) ) ;
```

In Scala: Case Classes

To create Datasets in Scala, you define a Scala case cl ass. A case cl ass is a regular class that has the following characteristics:

- Immutable
- Decomposable through pattern matching
- Allows for comparison based on structure instead of reference
- Easy to use and manipulate

These traits make it rather valuable for data analysis because it is quite easy to reason about a case class. Probably the most important feature is that case classes are immutable and allow for comparison by structure instead of value.

Here's how the Scala documentation describes it:

- Immutability frees you from needing to keep track of where and when things are mutated
- Comparison-by-value allows you to compare instances as if they were primitive values —no more uncertainty regarding whether instances of a class are compared by value or reference
- Pattern matching simplifies branching logic, which leads to less bugs and more readable code.

These advantages carry over to their usage within Spark, as well.

To begin creating a Dataset, let's define a case cl ass for one of our datasets:

```
case cl ass Fl i ght ( DEST_COUNTRY_NAME: St ri ng,
                      ORI GI N_COUNTRY_NAME: St ri ng, count : Bi gI nt )
```

Now that we defined a case cl ass, this will represent a single record in our dataset. More succintly, we now have a Dataset of Flights. This doesn't define any methods for us, simply the schema. When we read in our data, we'll get a DataFrame. However, we simply use the as method to cast it to our specified row type:

```
val f l i ght sDF = spark. read
```

```
. parquet ( "/dat a/f l i ght - dat a/parquet /2010- summary. parquet /") val f l i
ght s = f l i ght sDF. as[ Fl i ght ]
```

Actions

Even though we can see the power of Datasets, what's important to understand is that actions like col l ect , t ake, and count apply to whether we are using Datasets or DataFrames: `f l i ght s. show( 2)`

```
+-----------------+-------------------+-----+ |DEST_COUNTRY_NAME|ORI
GI N_COUNTRY_NAME|count |
+-----------------+-------------------+-----+
|   Uni t ed St at es|         Romani a|    1|
|   Uni t ed St at es|         I rel and|  264|
+-----------------+-------------------+-----+
```

You'll also notice that when we actually go to access one of the case cl asses, we don't need to do any type coercion, we simply specify the named attribute of the case cl ass and get back, not just the expected value but the expected type, as well: `f l i ght s. f i rst . DEST_COUNTRY_NAME // Uni t ed St at es`

Transformations

Transformations on Datasets are the same as those that we saw on DataFrames. Any transformation that you read about in this section is valid on a Dataset, and we encourage you to look through the specific sections on relevant aggregations or joins.

In addition to those transformations, Datasets allow us to specify more complex and strongly typed transformations than we could perform on DataFrames alone because we manipulate raw Java Virtual Machine (JVM) types. To illustrate this raw object manipulation, let's filter the Dataset that you just created.

Filtering

Let's look at a simple example by creating a simple function that accepts a Flight and returns a Boolean value that describes whether the origin and destination are the same. This is not a UDF (at least, in the way that Spark SQL defines UDF) but a generic function.

TIP

You'll notice in the following example that we're going to create a *function* to define this filter. This is an important difference from what we have done thus far in the book. By specifying a function, we are *forcing* Spark to evaluate this function on every row in our Dataset. This can be very resource intensive. For simple

filters it is always preferred to write SQL expressions. This will greatly reduce the cost of filtering out the data while still allowing you to manipulate it as a Dataset later on:

```
def originIsDestination(flight_row: Flight): Boolean = {
  return flight_row.ORIGIN_COUNTRY_NAME == flight_row.DEST_COUNTRY_NAME
}
```

We can now pass this function into the filter method specifying that for each row it should verify that this function returns true and in the process will filter our Dataset down accordingly: `flights.filter(flight_row => originIsDestination(flight_row)).first()` The result is:

```
Flight = Flight(United States, United States, 348113)
```

As we saw earlier, this function does not need to execute in Spark code at all. Similar to our UDFs, we can use it and test it on data on our local machines before using it within Spark.

For example, this dataset is small enough for us to collect to the driver (as an Array of Flights) on which we can operate and perform the exact same filtering operation: `flights.collect().filter(flight_row => originIsDestination(flight_row))` The result is:

```
Array[Flight] = Array(Flight(United States, United States, 348113))
```

We can see that we get the exact same answer as before.

Mapping

Filtering is a simple transformation, but sometimes you need to map one value to another value. We did this with our function in the previous example: it accepts a flight and returns a Boolean, but other times we might actually need to perform something more sophisticated like extract a value, compare a set of values, or something similar.

The simplest example is manipulating our Dataset such that we extract one value from each row.

This is effectively performing a DataFrame like select on our Dataset. Let's extract the destination: `val destinations = flights.map(f => f.DEST_COUNTRY_NAME)`

Notice that we end up with a Dataset of type String. That is because Spark already knows the JVM type that this result should return and allows us to benefit from compile-time checking if, for some reason, it is invalid.

We can collect this and get back an array of strings on the driver:

```
val localDestinations = destinations.take(5)
```

This might feel trivial and unnecessary; we can do the majority of this right on DataFrames. We in fact recommend that you do this because you gain so many benefits from doing so. You will gain advantages like code generation that are simply not possible with arbitrary user-defined functions. However, this can come in handy with much more sophisticated row-by-row manipulation.

Joins

Joins, as we covered earlier, apply just the same as they did for DataFrames. However Datasets also provide a more sophisticated method, the j oi nWi t h method. j oi nWi t h is roughly equal to a co-group (in RDD terminology) and you basically end up with two nested Datasets inside of one. Each column represents one Dataset and these can be manipulated accordingly. This can be useful when you need to maintain more information in the join or perform some more sophisticated manipulation on the entire result, like an advanced map or filter.

Let's create a fake flight metadata dataset to demonstrate j oi nWi t h:

```
case cl ass Fl i ght Met adat a( count : Bi gI nt , randomDat a: Bi gI nt )

val f l i ght sMet a = spark. range( 500) . map( x => ( x, scal a. ut i l . Random. next Long) )
  . wi t hCol umnRenamed( "_1", "count ") . wi t hCol umnRenamed( "_2", "randomDat a")
  . as[ Fl i ght Met adat a]

val f l i ght s2 = f l i ght s
  . j oi nWi t h( f l i ght sMet a, f l i ght s. col ( "count ") === f l i ght sMet a. col ( "count ") )
```

Notice that we end up with a Dataset of a sort of key-value pair, in which each row represents a Flight and the Flight Metadata. We can, of course, query these as a Dataset or a DataFrame with complex types: f l i ght s2. sel ect Expr("_1. DEST_COUNTRY_NAME") We can collect them just as we did before: f l i ght s2. t ake(2)

```
Array[ ( Fl i ght , Fl i ght Met adat a) ] = Array( ( Fl i ght ( Uni t ed St at es, Romani a, 1) , . . .
```

Of course, a "regular" join would work quite well, too, although you'll notice in this case that we end up with a DataFrame (and thus lose our JVM type information). val f l i ght s2 = f l i ght s. j oi n(f l i ght sMet a, Seq("count "))

We can always define another Dataset to gain this back. It's also important to note that there are no problems joining a DataFrame and a Dataset—we end up with the same result: val f l i ght s2 = f l i ght s. j oi n(f l i ght sMet a. t oDF( ) , Seq("count "))

Grouping and Aggregations

Grouping and aggregations follow the same fundamental standards that we saw in the previous aggregation chapter, so groupBy rol l up and cube still apply, but these return DataFrames instead of Datasets (you lose type information): f l i ght s. groupBy("DEST_COUNTRY_NAME") . count ( )

This often is not too big of a deal, but if you want to keep type information around there are other groupings and aggregations that you can perform. An excellent example is the groupByKey method. This allows you to group by a specific key in the Dataset and get a typed Dataset in return. This function, however, doesn't accept a specific column name but rather a function. This makes it possible for you to specify more sophisticated grouping functions that are much more akin to something like this: f l i ght s. groupByKey(x => x. DEST_COUNTRY_NAME) . count ( )

Although this provides flexibility, it's a trade-off because now we are introducing JVM types as well as functions that cannot be optimized by Spark. This means that you will see a performance difference and we can see this when we inspect the explain plan. In the following, you can see that we are effectivelly appending a new column to the DataFrame (the result of our function) and then performing the grouping on that: f l i ght s. groupByKey(x => x. DEST_COUNTRY_NAME) . count ( ) . expl ai n

```
== Physi cal Pl an ==
*HashAggregat e( keys=[ val ue#1396] , f unct i ons=[ count ( 1) ] )
+- Exchange hashpart i t i oni ng( val ue#1396, 200)
  +- *HashAggregat e( keys=[ val ue#1396] , f unct i ons=[ part i al _count ( 1) ] )
    +- *Proj ect [ val ue#1396]
      +- AppendCol umns <f unct i on1>, newI nst ance( cl ass . . .
      [ st at i ci nvoke( cl ass org. apache. spark. unsaf e. t ypes. UTF8St ri ng, . . .
        +- *Fi l eScan parquet [ D. . .
```

After we perform a grouping with a key on a Dataset, we can operate on the Key Value Dataset with functions that will manipulate the groupings as raw objects:

```
def grpSum( count ryName: St ri ng, val ues: I t erat or[ Fl i ght ] ) = {   val ues.
dropWhi l e( _. count < 5) . map( x => ( count ryName, x) )
}
f l i ght s. groupByKey( x => x. DEST_COUNTRY_NAME) . f l at MapGroups( grpSum) . show( 5)
+- - - - - - - - +- - - - - - - - - - - - - - - - - - +
|    _1|            _2|
+- - - - - - - - +- - - - - - - - - - - - - - - - - - +
|Angui l l a|[ Angui l l a, Uni t ed . . . |
```

```
|Paraguay|[ Paraguay, Uni t ed . . . |
|  Russi a|[ Russi a, Uni t ed St . . . |
| Senegal |[ Senegal , Uni t ed S. . . |
|  Sweden|[ Sweden, Uni t ed St . . . | +- -
- - - - - - +- - - - - - - - - - - - - - - - - - - +
```

```
def grpSum2( f : Fl i ght ) : I nt eger = {
  1
}
f l i ght s. groupByKey( x => x. DEST_COUNTRY_NAME) . mapVal ues( grpSum2) . count ( ) . t ake( 5)
```

We can even create new manipulations and define how groups should be reduced:

```
def sum2( l ef t : Fl i ght , ri ght : Fl i ght ) = {
  Fl i ght ( l ef t . DEST_COUNTRY_NAME, nul l , l ef t . count + ri ght . count )
}
f l i ght s. groupByKey( x => x. DEST_COUNTRY_NAME) . reduceGroups( ( l , r) => sum2( l , r) )  . t ake(
5)
```

It should be straightfoward enough to understand that this is a more expensive process than aggregating immediately after scanning, especially because it ends up in the same end result:

```
f l i ght s. groupBy( "DEST_COUNTRY_NAME") . count ( ) . expl ai n
```

```
== Physi cal Pl an ==
*HashAggregat e( keys=[ DEST_COUNTRY_NAME#1308] , f unct i ons=[ count ( 1) ] )
+- Exchange hashpart i t i oni ng( DEST_COUNTRY_NAME#1308, 200)
  +- *HashAggregat e( keys=[ DEST_COUNTRY_NAME#1308] , f unct i ons=[ part i al _count ( 1) ] )      +-
*Fi l eScan parquet [ DEST_COUNTRY_NAME#1308] Bat ched: t ru. . .
```

This should motivate using Datasets only with user-defined encoding surgically and only where it makes sense. This might be at the beginning of a big data pipeline or at the end of one.

Conclusion

In this chapter, we covered the basics of Datasets and provided some motivating examples. Although short, this chapter actually teaches you basically all that you need to know about Datasets and how to use them. It can be helpful to think of them as a blend between the higherlevel Structured APIs and the low-level RDD APIs, which is the topic of Chapter 12.

Part III. Low-Level APIs

Chapter 12. Resilient Distributed Datasets (RDDs)

The previous part of the book covered Spark's Structured APIs. You should heavily favor these APIs in almost all scenarios. That being said, there are times when higher-level manipulation will not meet the business or engineering problem you are trying to solve. For those cases, you might need to use Spark's lower-level APIs, specifically the Resilient Distributed Dataset (RDD), the SparkContext, and distributed *shared variables* like accumulators and broadcast variables. The chapters that follow in this part cover these APIs and how to use them.

> **WARNING**
>
> If you are brand new to Spark, this is not the place to start. Start with the Structured APIs, you'll be more productive more quickly!

What Are the Low-Level APIs?

There are two sets of low-level APIs: there is one for manipulating distributed data (RDDs), and another for distributing and manipulating distributed shared variables (broadcast variables and accumulators).

When to Use the Low-Level APIs?

You should generally use the lower-level APIs in three situations:

- You need some functionality that you cannot find in the higher-level APIs; for example, if you need very tight control over physical data placement across the cluster.
- You need to maintain some legacy codebase written using RDDs.
- You need to do some custom shared variable manipulation. We will discuss shared variables more in Chapter 14.

Those are the reasons why you should *use* these lower-level tools, buts it's still helpful to *understand* these tools because all Spark workloads compile down to these fundamental primitives. When you're calling a DataFrame transformation, it actually just becomes a set of RDD transformations. This understanding can make your task easier as you begin debugging more and more complex workloads.

Even if you are an advanced developer hoping to get the most out of Spark, we still recommend focusing on the Structured APIs. However, there are times when you might want to "drop down" to some of the lower-level tools to complete your task. You might need to drop down to these APIs to use some legacy code, implement some custom partitioner, or update and track the value of a variable over the course of a data pipeline's execution. These tools give you more finegrained control at the expense of safeguarding you from shooting yourself in the foot.

How to Use the Low-Level APIs?

A SparkCont ext is the entry point for low-level API functionality. You access it through the SparkSessi on, which is the tool you use to perform computation across a Spark cluster. We discuss this further in Chapter 15 but for now, you simply need to know that you can access a SparkContext via the following call: spark. sparkCont ext

About RDDs

RDDs were the primary API in the Spark 1.X series and are still available in 2.X, but they are not as commonly used. However, as we've pointed out earlier in this book, virtually all Spark code you run, whether DataFrames or Datasets, compiles down to an RDD. The Spark UI, covered in the next part of the book, also describes job execution in terms of RDDs. Therefore, it will behoove you to have at least a basic understanding of what an RDD is and how to use it.

In short, an RDD represents an immutable, partitioned collection of records that can be operated on in parallel. Unlike DataFrames though, where each record is a structured row containing fields with a known schema, in RDDs the records are just Java, Scala, or Python objects of the programmer's choosing.

RDDs give you complete control because every record in an RDD is a just a Java or Python object. You can store anything you want in these objects, in any format you want. This gives you great power, but not without potential issues. Every manipulation and interaction between values must be defined by hand, meaning that you must "reinvent the wheel" for whatever task you are trying to carry out. Also, optimizations are going to require much more manual work, because
Spark does not understand the inner structure of your records as it does with the Structured APIs. For instance, Spark's Structured APIs automatically store data in an optimzied, compressed binary format, so to achieve the same space-efficiency and performance, you'd also need to implement this type of format inside your objects and all the low-level operations to compute over it. Likewise, optimizations like reordering filters and aggregations that occur automatically in Spark SQL need to be implemented by hand. For this reason and others, we highly recommend using the Spark Structured APIs when possible.

The RDD API is similar to the Dat aset , which we saw in the previous part of the book, except that RDDs are not stored in, or manipulated with, the structured data engine. However, it is

trivial to convert back and forth between RDDs and Datasets, so you can use both APIs to take advantage of each API's strengths and weaknesses. We'll show how to do this throughout this part of the book.

Types of RDDs

If you look through Spark's API documentation, you will notice that there are lots of subclasses of RDD. For the most part, these are internal representations that the DataFrame API uses to create optimized physical execution plans. As a user, however, you will likely only be creating two types of RDDs: the "generic" RDD type or a key-value RDD that provides additional functions, such as aggregating by key. For your purposes, these will be the only two types of RDDs that matter. Both just represent a collection of objects, but key-value RDDs have special operations as well as a concept of custom partitioning by key.

Let's formally define RDDs. Internally, each RDD is characterized by five main properties:

- A list of partitions
- A function for computing each split
- A list of dependencies on other RDDs
- Optionally, a Part i t i oner for key-value RDDs (e.g., to say that the RDD is hashpartitioned)
- Optionally, a list of preferred locations on which to compute each split (e.g., block locations for a Hadoop Distributed File System [HDFS] file)

NOTE

The Part i t i oner is probably one of the core reasons why you might want to use RDDs in your code. Specifying your own custom Part i t i oner can give you significant performance and stability improvements if you use it correctly. This is discussed in more depth in Chapter 13 when we introduce Key–Value Pair RDDs.

These properties determine all of Spark's ability to schedule and execute the user program. Different kinds of RDDs implement their own versions of each of the aforementioned properties, allowing you to define new data sources.

RDDs follow the exact same Spark programming paradigms that we saw in earlier chapters. They provide *transformations*, which evaluate lazily, and *actions*, which evaluate eagerly, to manipulate data in a distributed fashion. These work the same way as transformations and actions on DataFrames and Datasets. However, there is no concept of "rows" in RDDs; individual records are just raw Java/Scala/Python objects, and you manipulate those manually instead of tapping into the repository of functions that you have in the structured APIs.

The RDD APIs are available in Python as well as Scala and Java. For Scala and Java, the performance is for the most part the same, the large costs incurred in manipulating the raw objects. Python, however, can lose a substantial amount of performance when using RDDs. Running Python RDDs equates to running Python user-defined functions (UDFs) row by row. Just as we saw in Chapter 6. We serialize the data to the Python process, operate on it in Python, and then serialize it back to the Java Virtual Machine (JVM). This causes a high overhead for Python RDD manipulations. Even though many people ran production code with them in the past, we recommend building on the Structured APIs in Python and only dropping down to RDDs if absolutely necessary.

When to Use RDDs?

In general, you should not manually create RDDs unless you have a very, very specific reason for doing so. They are a much lower-level API that provides a lot of power but also lacks a lot of the optimizations that are available in the Structured APIs. For the vast majority of use cases, DataFrames will be more efficient, more stable, and more expressive than RDDs.

The most likely reason for why you'll want to use RDDs is because you need fine-grained control over the physical distribution of data (custom partitioning of data).

Datasets and RDDs of Case Classes

We noticed this question on the web and found it to be an interesting one: what is the difference between RDDs of Case Classes and Datasets? The difference is that Datasets can still take advantage of the wealth of functions and optimizations that the Structured APIs have to offer. With Datasets, you do not need to choose between only operating on JVM types or on Spark types, you can choose whatever is either easiest to do or most flexible. You get the both of best worlds.

Creating RDDs

Now that we discussed some key RDD properties, let's begin applying them so that you can better understand how to use them.

Interoperating Between DataFrames, Datasets, and RDDs

One of the easiest ways to get RDDs is from an existing DataFrame or Dataset. Converting these to an RDD is simple: just use the rdd method on any of these data types. You'll notice that if you do a conversion from a Dataset[T] to an RDD, you'll get the appropriate native type T back (remember this applies only to Scala and Java):

```
// i n Scal a: convert s a Dat aset [ Long] t o RDD[ Long] spark. range(
500) . rdd
```

Because Python doesn't have Datasets—it has only DataFrames—you will get an RDD of type Row:

```
# in Python spark.range(10).rdd
```

To operate on this data, you will need to convert this Row object to the correct data type or extract values out of it, as shown in the example that follows. This is now an RDD of type Row:

```
// in Scala spark.range(10).toDF().rdd.map(rowObject => rowObject.getLong(0))

# in Python
spark.range(10).toDF("id").rdd.map(lambda row: row[0])
```

You can use the same methodology to create a DataFrame or Dataset from an RDD. All you need to do is call the toDF method on the RDD:

```
// in Scala spark.range(10).rdd.toDF()

# in Python
spark.range(10).rdd.toDF()
```

This command creates an RDD of type Row. This row is the internal Catalyst format that Spark uses to represent data in the Structured APIs. This functionality makes it possible for you to jump between the Structured and low-level APIs as it suits your use case. (We talk about this in Chapter 13.)

The RDD API will feel quite similar to the Dataset API in Chapter 11 because they are extremely similar to each other (RDDs being a lower-level representation of Datasets) that do not have a lot of the convenient functionality and interfaces that the Structured APIs do.

From a Local Collection

To create an RDD from a collection, you will need to use the parallelize method on a SparkContext (within a SparkSession). This turns a single node collection into a parallel collection. When creating this parallel collection, you can also explicitly state the number of partitions into which you would like to distribute this array. In this case, we are creating two partitions:

```
// in Scala val myCollection = "Spark The Definitive Guide : Big Data Processing Made Simple"
  .split(" ") val words = spark.sparkContext.parallelize(myCollection,
2)
```

```
# i n Pyt hon myCol l ect i on = "Spark The Def i ni t i ve Gui de : Bi g Dat a Processi ng Made Si mpl
e"\
  . spl i t ( " ") words = spark. sparkCont ext . paral l el i ze( myCol l ect
i on, 2)
```

An additional feature is that you can then name this RDD to show up in the Spark UI according to a given name:

```
// i n Scal a words. set Name(
"myWords") words. name //
myWords

# i n Pyt hon words. set Name(
"myWords") words. name( ) #
myWords
```

From Data Sources

Although you can create RDDs from data sources or text files, it's often preferable to use the Data Source APIs. RDDs do not have a notion of "Data Source APIs" like DataFrames do; they primarily define their dependency structures and lists of partitions. The Data Source API that we saw in Chapter 9 is almost always a better way to read in data. That being said, you can also read data as RDDs using sparkCont ext . For example, let's read a text file line by line: `spark. sparkCont ext . t ext Fi l e( "/some/pat h/wi t hText Fi l es")`

This creates an RDD for which each record in the RDD represents a line in that text file or files. Alternatively, you can read in data for which each text file should become a single record. The use case here would be where each file is a file that consists of a large JSON object or some document that you will operate on as an individual: `spark. sparkCont ext . whol eText Fi l es( "/some/pat h/wi t hText Fi l es")`

In this RDD, the name of the file is the first object and the value of the text file is the second string object.

Manipulating RDDs

You manipulate RDDs in much the same way that you manipulate DataFrames. As mentioned, the core difference being that you manipulate raw Java or Scala objects instead of Spark types. There is also a dearth of "helper" methods or functions that you can draw upon to simplify calculations. Rather, you must define each filter, map functions, aggregation, and any other manipulation that you want as a function.

To demonstrate some data manipulation, let's use the simple RDD (words) we created previously to define some more details.

Transformations

For the most part, many transformations mirror the functionality that you find in the Structured APIs. Just as you do with DataFrames and Datasets, you specify *transformations* on one RDD to create another. In doing so, we define an RDD as a dependency to another along with some manipulation of the data contained in that RDD.

distinct

A di st i nct method call on an RDD removes duplicates from the RDD:

words. di st i nct ( ) . count ( ) This gives a result of 10. **filter**

Filtering is equivalent to creating a SQL-like where clause. You can look through our records in the RDD and see which ones match some predicate function. This function just needs to return a Boolean type to be used as a filter function. The input should be whatever your given row is. In this next example, we filter the RDD to keep only the words that begin with the letter "S":

```
// i n Scal a def st art sWi t hS( i ndi vi dual : St ri
ng) = {   i ndi vi dual . st art sWi t h( "S")
}

# i n Pyt hon def st art sWi t hS( i ndi vi dual
) :   ret urn i ndi vi dual . st art swi t h( "S")
```

Now that we defined the function, let's filter the data. This should feel quite familiar if you read Chapter 11 because we simply use a function that operates record by record in the RDD. The function is defined to work on each record in the RDD individually:

```
// i n Scal a words. f i l t er( word => st art sWi t hS( word) ) .
col l ect ( )

# i n Pyt hon
words. f i l t er( l ambda word: st art sWi t hS( word) ) . col l ect ( )
```

This gives a result of *Spark* and *Simple*. We can see, like the Dataset API, that this returns native types. That is because we never coerce our data into type Row, nor do we need to convert the data after collecting it.

map

Mapping is again the same operation that you can read about in Chapter 11. You specify a function that returns the value that you want, given the correct input. You then apply that, record by record. Let's perform something similar to what we just did. In this example, we'll map the current word to the word, its starting letter, and whether the word begins with "S."

Notice in this instance that we define our functions completely inline using the relevant lambda syntax:

```
// in Scala val words2 = words.map(word => (word, word(0), word.startsWith(
"S")))

# in Python
words2 = words.map(lambda word: (word, word[0], word.startswith("S")))
```

You can subsequently filter on this by selecting the relevant Boolean value in a new function:

```
// in Scala words2.filter(record => record._3).t
ake(5)

# in Python
words2.filter(lambda record: record[2]).take(5)
```

This returns a tuple of "Spark," "S," and "true," as well as "Simple," "S," and "True."

flatMap

flatMap provides a simple extension of the map function we just looked at. Sometimes, each current row should return multiple rows, instead. For example, you might want to take your set of words and flatMap it into a set of characters. Because each word has multiple characters, you should use flatMap to expand it. flatMap requires that the ouput of the map function be an iterable that can be expanded:

```
// in Scala words.flatMap(word => word.toSeq)
.take(5)

# in Python
words.flatMap(lambda word: list(word)).take(5) This
```

yields *S*, *P*, *A*, *R*, *K*.

sort

To sort an RDD you must use the sortBy method, and just like any other RDD operation, you do this by specifying a function to extract a value from the objects in your RDDs and then sort

based on that. For instance, the following example sorts by word length from longest to shortest:

```
// in Scala
words.sortBy(word => word.length() * -1).take(2)

# in Python
words.sortBy(lambda word: len(word) * -1).take(2)
```

Random Splits

We can also randomly split an RDD into an Array of RDDs by using the randomSplit method, which accepts an Array of weights and a random seed:

```
// in Scala
val fiftyFiftySplit = words.randomSplit(Array[Double](0.5, 0.5))

# in Python
fiftyFiftySplit = words.randomSplit([0.5, 0.5])
```

This returns an array of RDDs that you can manipulate individually.

Actions

Just as we do with DataFrames and Datasets, we specify *actions* to kick off our specified transformations. Actions either collect data to the driver or write to an external data source.

reduce

You can use the reduce method to specify a function to "reduce" an RDD of any kind of value to one value. For instance, given a set of numbers, you can reduce this to its sum by specifying a function that takes as input two values and reduces them into one. If you have experience in functional programming, this should not be a new concept:

```
// in Scala spark.sparkContext.parallelize(1 to 20).reduce(_ + _) //
210

# in Python
spark.sparkContext.parallelize(range(1, 21)).reduce(lambda x, y: x + y) # 210
```

You can also use this to get something like the longest word in our set of words that we defined a moment ago. The key is just to define the correct function:

```
// in Scala def wordLengthReducer(leftWord: String, rightWord: String): Stri
ng = {  if (leftWord.length > rightWord.length)   return leftWord  else   ret
urn rightWord
}
words.reduce(wordLengthReducer)

# in Python def wordLengthReducer(leftWord, ri
ght Word) :  if len(leftWord) > len(rightWord) :
  return leftWord  else:   return ri
ght  Word  words.  reduce(  wordLengt
hReducer)
```

This reducer is a good example because you can get one of two outputs. Because the reduce operation on the partitions is not deterministic, you can have either “definitive” or “processing” (both of length 10) as the “left” word. This means that sometimes you can end up with one, whereas other times you end up with the other.

count

This method is fairly self-explanatory. Using it, you could, for example, count the number of rows in the RDD: `words.count()`

countApprox

Even though the return signature for this type is a bit strange, it’s quite sophisticated. This is an approximation of the count method we just looked at, but it must execute within a timeout (and can return incomplete results if it exceeds the timeout).

The confidence is the probability that the error bounds of the result will contain the true value. That is, if countApprox were called repeatedly with confidence 0.9, we would expect 90% of the results to contain the true count. The confidence must be in the range [0,1], or an exception will be thrown:

```
val confidence = 0.95 val timeout
Millis seconds = 400
words.countApprox(timeout Millis, confidence) countApproxDistinct
```

There are two implementations of this, both based on streamlib’s implementation of “HyperLogLog in Practice: Algorithmic Engineering of a State-of-the-Art Cardinality Estimation Algorithm.”

In the first implementation, the argument we pass into the function is the relative accuracy. Smaller values create counters that require more space. The value must be greater than 0.000017:

```
words. count ApproxDi st i nct ( 0. 05)
```

With the other implementation you have a bit more control; you specify the relative accuracy based on two parameters: one for “regular” data and another for a sparse representation.

The two arguments are p and sp where p is precision and sp is sparse precision. The relative accuracy is approximately 1. 054 / sqrt (2^P) . Setting a nonzero (sp > p) can reduce the memory consumption and increase accuracy when the cardinality is small. Both values are integers: words. count ApproxDi st i nct (4, 10)

countByValue

This method counts the number of values in a given RDD. However, it does so by finally loading the result set into the memory of the driver. You should use this method only if the resulting map is expected to be small because the entire thing is loaded into the driver’s memory. Thus, this method makes sense only in a scenario in which either the total number of rows is low or the number of distinct items is low: words. count ByVal ue( )

countByValueApprox

This does the same thing as the previous function, but it does so as an approximation. This must execute within the specified timeout (first parameter) (and can return incomplete results if it exceeds the timeout).

The confidence is the probability that the error bounds of the result will contain the true value. That is, if count Approx were called repeatedly with confidence 0. 9, we would expect 90% of the results to contain the true count. The confidence must be in the range [0,1], or an exception will be thrown: words. count ByVal ueApprox(1000, 0. 95) **first**

The f i rst method returns the first value in the dataset:

```
words. f i rst ( )
```

max and min

max and mi n return the maximum values and minimum values, respectively:

```
spark. sparkCont ext . paral l el i ze( 1 t o 20) . max( )
```

```
spark. sparkCont ext . paral l el i ze( 1 t o 20) . mi n( )
```

take

t ake and its derivative methods take a number of values from your RDD. This works by first scanning one partition and then using the results from that partition to estimate the number of additional partitions needed to satisfy the limit.

There are many variations on this function, such as t akeOrdered, t akeSampl e, and t op. You can use t akeSampl e to specify a fixed-size random sample from your RDD. You can specify whether this should be done by using wi t hRepl acement , the number of values, as well as the random seed. t op is effectively the opposite of t akeOrdered in that it selects the top values according to the implicit ordering:

```
words.  t  ake(  5)  words.  t
akeOrdered(  5)  words.  t  op(  5)
val wi t hRepl acement = t rue val
numberToTake     =     6     val
randomSeed = 100L
words. t akeSampl e( wi t hRepl acement , numberToTake, randomSeed)
```

Saving Files

Saving files means writing to plain-text files. With RDDs, you cannot actually “save” to a data source in the conventional sense. You must iterate over the partitions in order to save the contents of each partition to some external database. This is a low-level approach that reveals the underlying operation that is being performed in the higher-level APIs. Spark will take each partition, and write that out to the destination.

saveAsTextFile

To save to a text file, you just specify a path and optionally a compression codec:

```
words. saveAsText Fi l e( "f i l e: /t mp/bookTi t l e")
```

To set a compression codec, we must import the proper codec from Hadoop. You can find these in the org. apache. hadoop. i o. compress library:

```
// i n Scal a i mport org. apache. hadoop. i o. compress. BZi
p2Codec
words. saveAsText Fi l e( "f i l e: /t mp/bookTi t l eCompressed", cl assOf [ BZi p2Codec] )
```

SequenceFiles

Spark originally grew out of the Hadoop ecosystem, so it has a fairly tight integration with a variety of Hadoop tools. A sequenceFi l e is a flat file consisting of binary key–value pairs. It is extensively used in MapReduce as input/output formats.

Spark can write to sequenceFi l es using the saveAsObj ect Fi l e method or by explicitly writing key–value pairs, as described in Chapter 13: words. saveAsObj ect Fi l e("/t mp/my/sequenceFi l ePat h")

Hadoop Files

There are a variety of different Hadoop file formats to which you can save. These allow you to specify classes, output formats, Hadoop configurations, and compression schemes. (For information on these formats, read *Hadoop: The Definitive Guide* [O'Reilly, 2015].) These formats are largely irrelevant except if you're working deeply in the Hadoop ecosystem or with some legacy mapReduce jobs.

Caching

The same principles apply for caching RDDs as for DataFrames and Datasets. You can either cache or persist an RDD. By default, cache and persist only handle data in memory. We can name it if we use the set Name function that we referenced previously in this chapter: words. cache( )

We can specify a storage level as any of the storage levels in the singleton object: org. apache. spark. st orage. St orageLevel , which are combinations of memory only; disk only; and separately, off heap.

We can subsequently query for this storage level (we talk about storage levels when we discuss persistence in Chapter 20):

```
// i n Scal a words. get St
orageLevel

# i n Pyt hon words. get St
orageLevel ( )
```

Checkpointing

One feature not available in the DataFrame API is the concept of *checkpointing*. Checkpointing is the act of saving an RDD to disk so that future references to this RDD point to those intermediate partitions on disk rather than recomputing the RDD from its original source. This is similar to caching except that it's not stored in memory, only disk. This can be helpful when performing iterative computation, similar to the use cases for caching:

```
spark. sparkCont ext . set Checkpoi nt Di r( "/some/pat h/f or/checkpoi nt i ng") words.
checkpoi nt ( )
```

Now, when we reference this RDD, it will derive from the checkpoint instead of the source data. This can be a helpful optimization.

Pipe RDDs to System Commands

The pi pe method is probably one of Spark's more interesting methods. With pipe, you can return an RDD created by piping elements to a forked external process. The resulting RDD is computed by executing the given process once per partition. All elements of each input partition are written to a process's stdin as lines of input separated by a newline. The resulting partition consists of the process's stdout output, with each line of stdout resulting in one element of the output partition. A process is invoked even for empty partitions.

The print behavior can be customized by providing two functions.

We can use a simple example and pipe each partition to the command wc. Each row will be passed in as a new line, so if we perform a line count, we will get the number of lines, one per partition: `words. pi pe( "wc - l ") . col l ect ( )`

In this case, we got five lines per partition.

mapPartitions

The previous command revealed that Spark operates on a per-partition basis when it comes to actually executing code. You also might have noticed earlier that the return signature of a map function on an RDD is actually MapPart i t i onsRDD. This is because map is just a row-wise alias for mapPart i t i ons, which makes it possible for you to map an individual partition (represented as an iterator). That's because physically on the cluster we operate on each partition individually (and not a specific row). A simple example creates the value "1" for every partition in our data, and the sum of the following expression will count the number of partitions we have:

```
// i n Scal a words. mapPart i t i ons( part => I t erat or[ I nt ] ( 1) ) . sum(
) // 2

# i n Pyt hon
words. mapPart i t i ons( l ambda part : [ 1] ) . sum( ) # 2
```

Naturally, this means that we operate on a per-partition basis and allows us to perform an operation on that *entire* partition. This is valuable for performing something on an entire subdataset of your RDD. You can gather all values of a partition class or group into one partition and then operate on that entire group using arbitrary functions and controls. An example use case of this would be that you could pipe this through some custom machine learning algorithm and train an individual model for that company's portion of the dataset. A Facebook engineer

has an interesting demonstration of their particular implementation of the pi pe operator with a similar use case demonstrated at Spark Summit East 2017.

Other functions similar to mapPart i t i ons include mapPart i t i onsWi t hI ndex. With this you specify a function that accepts an index (within the partition) and an iterator that goes through all items within the partition. The partition index is the partition number in your RDD, which identifies where each record in our dataset sits (and potentially allows you to debug). You might use this to test whether your map functions are behaving correctly:

```
// i n Scal a def i ndexedFunc( part i t i onI ndex: I nt , wi t hi nPart I t erat or: I t erat or[ St ri ng] )
= {  wi t hi nPart I t erat or. t oLi st . map(    val ue => s"Part i t i on: $part i t i onI ndex => $val ue")
. i t erat or
}
words. mapPart i t i onsWi t hI ndex( i ndexedFunc) . col l ect ( )

# i n Pyt hon def i ndexedFunc( part i t i onI ndex, wi t hi nPart I t erat
or) :   ret urn [ "part i t i on: { } => { } ". f ormat ( part i t i onI ndex,
x) f or x i n wi t hi nPart I t erat or]
words. mapPart i t i onsWi t hI ndex( i ndexedFunc) . col l ect ( )
```

foreachPartition

Although mapPart i t i ons needs a return value to work properly, this next function does not. f oreachPart i t i on simply iterates over all the partitions of the data. The difference is that the function has no return value. This makes it great for doing something with each partition like writing it out to a database. In fact, this is how many data source connectors are written. You can create our own text file source if you want by specifying outputs to the temp directory with a random ID:

```
words. f oreachPart i t i on { i t er =>  i mport j ava. i o. _  i mport scal a. ut i l . Random  val randomFi
l eName = new Random( ) . next I nt ( )  val pw = new Pri nt Wri t er( new Fi l e( s"/t mp/random- f i
l e- ${ randomFi l eName} . t xt ") )  whi l e ( i t er. hasNext ) {    pw. wri t e( i t er. next ( ) )
 }  pw. cl ose(
)
}
```

You'll find these two files if you scan your */tmp* directory.

glom

gl om is an interesting function that takes every partition in your dataset and converts them to arrays. This can be useful if you're going to collect the data to the driver and want to have an array for each partition. However, this can cause serious stability issues because if you have large partitions or a large number of partitions, it's simple to crash the driver.

In the following example, you can see that we get two partitions and each word falls into one partition each:

```
// in Scala spark.sparkContext.parallelize(Seq("Hello", "World"), 2).glom().collect() // Array(Array(Hello), Array(World))

# in Python spark.sparkContext.parallelize(["Hello", "World"], 2).glom().collect() # [['Hello'], ['World']]
```

Conclusion

In this chapter, you saw the basics of the RDD APIs, including single RDD manipulation. Chapter 13 touches on more advanced RDD concepts, such as joins and key-value RDDs.

Chapter 13. Advanced RDDs

Chapter 12 explored the basics of single RDD manipulation. You learned how to create RDDs and why you might want to use them. In addition, we discussed map, filter, reduce, and how to create functions to transform single RDD data. This chapter covers the advanced RDD operations and focuses on key–value RDDs, a powerful abstraction for manipulating data. We also touch on some more advanced topics like custom partitioning, a reason you might want to use RDDs in the first place. With a custom partitioning function, you can control exactly how data is laid out on the cluster and manipulate that individual partition accordingly. Before we get there, let's summarize the key topics we will cover:

- Aggregations and key–value RDDs
- Custom partitioning
- RDD joins

> **NOTE**
>
> This set of APIs has been around since, essentially, the beginning of Spark, and there are a *ton* of examples all across the web on this set of APIs. This makes it trivial to search and find examples that will show you how to use these operations.

Let's use the same dataset we used in the last chapter:

```
// in Scala val myCollection = "Spark The Definitive Guide : Big Data Processing Made Simple"
  .split(" ") val words = spark.sparkContext.parallelize( myCollect
on, 2)

# in Python myCollection = "Spark The Definitive Guide : Big Data Processing Made Simp
le"\
  .split(" ")
words = spark.sparkContext.parallelize( myCollection, 2)
```

Key-Value Basics (Key-Value RDDs)

There are many methods on RDDs that require you to put your data in a key–value format. A hint that this is required is that the method will include <some- operation>ByKey. Whenever you see ByKey in a method name, it means that you can perform this only on a PairRDD type. The easiest way is to just map over your current RDD to a basic key–value structure. This means having two values in each record of your RDD:

```
// in Scala words.map(word => (word.t
oLowerCase, 1))

# in Python
words.map(lambda word: (word.lower(), 1))
```

keyBy

The preceding example demonstrated a simple way to create a key. However, you can also use the keyBy function to achieve the same result by specifying a function that creates the key from your current value. In this case, you are keying by the first letter in the word. Spark then keeps the record as the value for the keyed RDD:

```
// in Scala
val keyword = words.keyBy(word => word.toLowerCase.toSeq(0).toString)

# in Python
keyword = words.keyBy(lambda word: word.lower()[0])
```

Mapping over Values

After you have a set of key–value pairs, you can begin manipulating them as such. If we have a tuple, Spark will assume that the first element is the key, and the second is the value. When in this format, you can explicitly choose to map-over the values (and ignore the individual keys). Of course, you could do this manually, but this can help prevent errors when you know that you are just going to modify the values:

```
// in Scala keyword.mapValues(word => word.toUpperCase).
collect()

# in Python
keyword.mapValues(lambda word: word.upper()).collect() Here's
```

the output in Python:

```
[('s','SPARK'),
('t','THE'),
('d','DEFINITIVE'),
('g','GUIDE'),
(':',':'),
('b','BIG'),
('d','DATA'),
('p','PROCESSING'),
('m','MADE'),
('s','SIMPLE')]
```

(The values in Scala are the same but omitted for brevity.)

You can f l at Map over the rows, as we saw in Chapter 12, to expand the number of rows that you have to make it so that each row represents a character. In the following example, we will omit the output, but it would simply be each character as we converted them into arrays:

```
// i n Scal a keyword. f l at MapVal ues( word => word. t oUpperCase) .
col l ect ( )

# i n Pyt hon
keyword. f l at MapVal ues( l ambda word: word. upper( ) ) . col l ect ( )
```

Extracting Keys and Values

When we are in the key–value pair format, we can also extract the specific keys or values by using the following methods:

```
// i n Scal a keyword. keys. col
l ect ( ) keyword. val ues. col l
ect ( )

# i n Pyt hon keyword. keys( ) .
col l ect ( ) keyword. val ues( ) .
col l ect ( )
```

lookup

One interesting task you might want to do with an RDD is look up the result for a particular key. Note that there is *no* enforcement mechanism with respect to there being only one key for each input, so if we l ookup “s”, we are going to get both values associated with that—“Spark” and “Simple”: keyword. l ookup("s")

sampleByKey

There are two ways to sample an RDD by a set of keys. We can do it via an approximation or exactly. Both operations can do so with or without replacement as well as sampling by a fraction by a given key. This is done via simple random sampling with one pass over the RDD, which produces a sample of size that’s approximately equal to the sum of mat h. cei l (numI t ems * sampl i ngRat e) over all key values:

```
// i n Scal a val di st i nct Chars = words. f l at Map( word => word. t oLowerCase. t oSeq) . di st
i nct
  . col l ect ( ) i mport scal a. ut i l . Random val sampl eMap = di st i nct Chars. map( c => ( c, new
Random( ) . next Doubl e( ) ) ) . t oMap words. map( word => ( word. t oLowerCase. t oSeq( 0) ,
word) )
```

```
  .sampleByKey(true, sampleMap, 6L)
  .collect()

# in Python import random distinct Chars = words.flatMap(lambda word: list(word.
lower())).distinct().\
  collect() sampleMap = dict(map(lambda c: (c, random.random()), distinct
Chars)) words.map(lambda word: (word.lower()[0], word)).\
  sampleByKey(True, sampleMap, 6).collect()
```

This method differs from sampleByKey in that you make additional passes over the RDD to create a sample size that's exactly equal to the sum of math.ceil(numItems * samplingRate) over all key values with a 99.99% confidence. When sampling without replacement, you need one additional pass over the RDD to guarantee sample size; when sampling with replacement, you need two additional passes:

```
// in Scala words.map(word => (word.toLowerCase.toSeq(0)
, word))  .sampleByKeyExact(true, sampleMap, 6L).collect(
)
```

Aggregations

You can perform aggregations on plain RDDs or on PairRDDs, depending on the method that you are using. Let's use some of our datasets to demonstrate this:

```
// in Scala val chars = words.flatMap(word => word.toLowerCase.t
oSeq) val KVcharacters = chars.map(letter => (letter, 1)) def
maxFunc(left:Int, right:Int) = math.max(left, right) def addFunc(
left:Int, right:Int) = left + right val nums = sc.parallelize(1 to
30, 5)

# in Python
chars = words.flatMap(lambda word: word.lower())
KVcharacters = chars.map(lambda letter: (letter, 1)) def
maxFunc(left, right): return max(left, right) def addFunc(
left, right): return left + right
nums = sc.parallelize(range(1, 31), 5)
```

After you have this, you can do something like countByKey, which counts the items per each key.

countByKey

You can count the number of elements for each key, collecting the results to a local Map. You can also do this with an approximation, which makes it possible for you to specify a timeout and confidence when using Scala or Java:

```
// in Scala val timeout = 1000L //milli i
seconds val confidence = 0.95
KVcharacters.countByKey()
KVcharacters.countByKeyApprox(timeout, confidence)

# in Python
KVcharacters.countByKey()
```

Understanding Aggregation Implementations

There are several ways to create your key–value PairRDDs; however, the implementation is actually quite important for job stability. Let's compare the two fundamental choices, groupBy and reduce. We'll do these in the context of a key, but the same basic principles apply to the groupBy and reduce methods.

groupByKey

Looking at the API documentation, you might think groupByKey with a map over each grouping is the best way to sum up the counts for each key:

```
// in Scala
KVcharacters.groupByKey().map(row => (row._1, row._2.reduce(addFunc))).collect()

# in Python
KVcharacters.groupByKey().map(lambda row: (row[0], reduce(addFunc, row[1]))) \
  .collect()
# not e this is Python 2, reduce must be imported from functools in Python 3
```

However, this is, for the majority of cases, the wrong way to approach the problem. The fundamental issue here is that each executor must hold *all values* for a given key in memory before applying the function to them. Why is this problematic? If you have massive key skew, some partitions might be completely overloaded with a ton of values for a given key, and you will get Out Of MemoryErrors. This obviously doesn't cause an issue with our current dataset, but it can cause serious problems at scale. This is not guaranteed to happen, but it *can* happen.

There are use cases when groupByKey does make sense. If you have consistent value sizes for each key and know that they will fit in the memory of a given executor, you're going to be just fine. It's just good to know exactly what you're getting yourself into when you do this. There is a preferred approach for additive use cases: reduceByKey.

reduceByKey

Because we are performing a simple count, a much more stable approach is to perform the same

flatMap and then just perform a map to map each letter instance to the number one, and then perform a reduceByKey with a summation function in order to collect back the array. This

implementation is much more stable because the reduce happens within each partition and doesn't need to put everything in memory. Additionally, there is no incurred shuffle during this operation; everything happens at each worker individually before performing a final reduce. This greatly enhances the speed at which you can perform the operation as well as the stability of the operation:

```
KVcharact ers. reduceByKey( addFunc) . col l ect ( )
```

Here's the result of the operation:

```
Array( ( d, 4) , ( p, 3) , ( t , 3) , ( b, 1) , ( h, 1) , ( n, 2) , . . .
( a, 4) , ( i , 7) , ( k, 1) , ( u, 1) , ( o, 1) , ( g, 3) , ( m, 2) , ( c, 1) )
```

The reduceByKey method returns an RDD of a group (the key) and sequence of elements that are not guranteed to have an ordering. Therefore this method is completely appropriate when our workload is associative but inappropriate when the order matters.

Other Aggregation Methods

There exist a number of advanced aggregation methods. For the most part these are largely implementation details depending on your specific workload. We find it very rare that users come across this sort of workload (or need to perform this kind of operation) in modern-day Spark. There just aren't that many reasons for using these extremely low-level tools when you can perform much simpler aggregations using the Structured APIs. These functions largely allow you very specific, very low-level control on exactly how a given aggregation is performed on the cluster of machines.

aggregate

Another function is aggregat e. This function requires a null and start value and then requires you to specify two different functions. The first aggregates within partitions, the second aggregates across partitions. The start value will be used at both aggregation levels:

```
// i n Scal a nums. aggregat e( 0) ( maxFunc,
addFunc)

# i n Pyt hon
nums. aggregat e( 0, maxFunc, addFunc)
```

aggregat e does have some performance implications because it performs the final aggregation on the driver. If the results from the executors are too large, they can take down the driver with an Out Of MemoryError. There is another method, t reeAggregat e that does the same thing as aggregate (at the user level) but does so in a different way. It basically "pushes down" some of the subaggregations (creating a tree from executor to executor) before performing the final

aggregation on the driver. Having multiple levels can help you to ensure that the driver does not run out of memory in the process of the aggregation. These tree-based implementations are often to try to improve stability in certain operations:

```
// i n Scal a val dept h = 3 nums. t reeAggregat e( 0) (
maxFunc, addFunc, dept h)

# i n Pyt hon dept
h = 3
nums. t reeAggregat e( 0, maxFunc, addFunc, dept h)
```

aggregateByKey

This function does the same as aggregat e but instead of doing it partition by partition, it does it by key. The start value and functions follow the same properties:

```
// i n Scal a
KVcharact ers. aggregat eByKey( 0) ( addFunc, maxFunc) . col l ect ( )

# i n Pyt hon
KVcharact ers. aggregat eByKey( 0, addFunc, maxFunc) . col l ect ( )
```

combineByKey

Instead of specifying an aggregation function, you can specify a combiner. This combiner operates on a given key and merges the values according to some function. It then goes to merge the different outputs of the combiners to give us our result. We can specify the number of output partitions as a custom output partitioner as well:

```
// i n Scal a val val ToCombi ner = ( val ue: I nt ) => Li st ( val ue) val mergeVal uesFunc = ( val s: Li st
[ I nt ] , val ToAppend: I nt ) => val ToAppend : : val s val mergeCombi nerFunc = ( val s1: Li st [ I nt ]
, val s2: Li st [ I nt ] ) => val s1 : : : val s2
// now we def i ne t hese as f unct i on vari abl es val out
put Part i t i ons = 6
KVcharact ers . combi
neByKey( val ToCombi
ner, mergeVal uesFunc,
mergeCombi nerFunc,
out put Part i t i ons) . col l
ect ( )

# i n Pyt hon def val ToCombi
ner( val ue) :
 ret urn [ val ue]
def mergeVal uesFunc( val s, val ToAppend) :
 val s. append( val ToAppend) ret urn val s
def mergeCombi nerFunc( val s1, val s2) :
```

```
 ret urn val s1 + val s2 out
put  Part  i  t  i  ons  =  6
KVcharact  ers\     .  combi
neByKey(   val ToCombi ner,
mergeVal           uesFunc,
mergeCombi nerFunc,   out
put Part i t i ons) \  . col l ect
( )
```

foldByKey

f ol dByKey merges the values for each key using an associative function and a neutral "zero value," which can be added to the result an arbitrary number of times, and must not change the result (e.g., 0 for addition, or 1 for multiplication):

```
// i n Scal a
KVcharact ers. f ol dByKey( 0) ( addFunc) . col l ect ( )

# i n Pyt hon
KVcharact ers. f ol dByKey( 0, addFunc) . col l ect ( )
```

CoGroups

CoGroups give you the ability to group together up to three key–value RDDs together in Scala and two in Python. This joins the given values by key. This is effectively just a group-based join on an RDD. When doing this, you can also specify a number of output partitions or a custom partitioning function to control exactly how this data is distributed across the cluster (we talk about partitioning functions later on in this chapter):

```
// i n Scal a i mport scal a. ut i l . Random val di st i nct Chars = words. f l at Map( word => word.
t oLowerCase. t oSeq) . di st i nct val charRDD = di st i nct Chars. map( c => ( c, new Random( )
. next Doubl e( ) ) ) val charRDD2 = di st i nct Chars. map( c => ( c, new Random( ) . next Doubl
e( ) ) ) val charRDD3 = di st i nct Chars. map( c => ( c, new Random( ) . next Doubl e( ) ) )
charRDD. cogroup( charRDD2, charRDD3) . t ake( 5)

# i n Pyt hon i mport random di st i nct Chars = words. f l at Map( l ambda word: word.
l ower( ) ) . di st i nct ( ) charRDD = di st i nct Chars. map( l ambda c: ( c, random.
random( ) ) ) charRDD2 = di st i nct Chars. map( l ambda c: ( c, random. random( ) ) )
charRDD. cogroup( charRDD2) . t ake( 5)
```

The result is a group with our key on one side, and all of the relevant values on the other side.

Joins

RDDs have much the same joins as we saw in the Structured API, although RDDs are much more involved for you. They all follow the same basic format: the two RDDs we would like to join,

and, optionally, either the number of output partitions or the customer partition function to which they should output. We'll talk about partitioning functions later on in this chapter.

Inner Join

We'll demonstrate an inner join now. Notice how we are setting the number of output partitions we would like to see:

```
// i n Scal a val keyedChars = di st i nct Chars. map( c => ( c, new Random( ) . next Doubl e(
) ) ) val out put Part i t i ons = 10
KVcharact ers. j oi n( keyedChars) . count ( )
KVcharact ers. j oi n( keyedChars, out put Part i t i ons) . count ( )

# i n Pyt hon keyedChars = di st i nct Chars. map( l ambda c: ( c, random.
random( ) ) ) out put Part i t i ons = 10
KVcharact ers. j oi n( keyedChars) . count ( )
KVcharact ers. j oi n( keyedChars, out put Part i t i ons) . count ( )
```

We won't provide an example for the other joins, but they all follow the same basic format. You can learn about the following join types at the conceptual level in Chapter 8:

- f ul l Out erJ oi n l
- ef t Out erJ oi n ri
- ght Out erJ oi n
- cart esi an (This, again, is very dangerous! It does not accept a join key and can have a massive output.)

zips

The final type of join isn't really a join at all, but it does combine two RDDs, so it's worth labeling it as a join. zi p allows you to "zip" together two RDDs, assuming that they have the same length. This creates a Pai rRDD. The two RDDs must have the same number of partitions as well as the same number of elements:

```
// i n Scal a val numRange = sc. paral l el i ze( 0 t o
9, 2) words. zi p( numRange) . col l ect ( )

# i n Pyt hon numRange = sc. paral l el i ze( range(
10) , 2) words. zi p( numRange) . col l ect ( )
```

This gives us the following result, an array of keys zipped to the values:

```
[ ( ' Spark' , 0) ,
 ( ' The' , 1) ,
```

```
( ' Def i ni t i ve' , 2) ,
( ' Gui de' , 3) ,
( ' : ' , 4) ,
( ' Bi g' , 5) ,
( ' Dat a' , 6) ,
( ' Processi ng' , 7) ,
( ' Made' , 8) ,
( ' Si mpl e' , 9) ]
```

Controlling Partitions

With RDDs, you have control over how data is exactly physically distributed across the cluster. Some of these methods are basically the same from what we have in the Structured APIs but the key addition (that does not exist in the Structured APIs) is the ability to specify a partitioning function (formally a custom Part i t i oner, which we discuss later when we look at basic methods).

coalesce

coal esce effectively collapses partitions on the same worker in order to avoid a shuffle of the data when repartitioning. For instance, our words RDD is currently two partitions, we can collapse that to one partition by using coal esce without bringing about a shuffle of the data:

```
// i n Scal a words. coal esce( 1) . get NumPart i t i
ons // 1

# i n Pyt hon
words. coal esce( 1) . get NumPart i t i ons( ) # 1
```

repartition

The repart i t i on operation allows you to repartition your data up or down but performs a shuffle across nodes in the process. Increasing the number of partitions can increase the level of parallelism when operating in map- and filter-type operations: `words. repart i t i on( 10) // gi ves us 10 part i t i ons`

repartitionAndSortWithinPartitions

This operation gives you the ability to repartition as well as specify the ordering of each one of those output partitions. We'll omit the example because the documentation for it is good, but both the partitioning and the key comparisons can be specified by the user.

Custom Partitioning

This ability is one of the primary reasons you'd want to use RDDs. Custom partitioners are not available in the Structured APIs because they don't really have a logical counterpart. They're a low-level, implementation detail that can have a significant effect on whether your jobs run successfully. The canonical example to motivate custom partition for this operation is PageRank whereby we seek to control the layout of the data on the cluster and avoid shuffles. In our shopping dataset, this might mean partitioning by each customer ID (we'll get to this example in a moment).

In short, the sole goal of custom partitioning is to even out the distribution of your data across the cluster so that you can work around problems like data skew.

If you're going to use custom partitioners, you should drop down to RDDs from the Structured APIs, apply your custom partitioner, and then convert it back to a DataFrame or Dataset. This way, you get the best of both worlds, only dropping down to custom partitioning when you need to.

To perform custom partitioning you need to implement your own class that extends Part i t i oner. You need to do this only when you have lots of domain knowledge about your problem space—if you're just looking to partition on a value or even a set of values (columns), it's worth just doing it in the DataFrame API.

Let's dive into an example:

```
// i n Scal a val df = spark. read. opt i on( "header", "t rue") . opt i on( "i nf erSchema", "t rue")
  . csv( "/dat a/ret ai l - dat a/al l /") val rdd
= df . coal esce( 10) . rdd

# i n Pyt hon df = spark. read. opt i on( "header", "t rue") . opt i on( "i nf erSchema", "t
rue") \
  . csv( "/dat a/ret ai l - dat a/al l /") rdd = df
. coal esce( 10) . rdd

df . pri nt Schema( )
```

Spark has two built-in Partitioners that you can leverage off in the RDD API, a HashPart i t i oner for discrete values and a RangePart i t i oner. These two work for discrete values and continuous values, respectively. Spark's Structured APIs will already use these, although we can use the same thing in RDDs:

```
// i n Scal a i mport org. apache. spark. HashPart i t i oner rdd. map( r => r( 6) ) . t
ake( 5) . f oreach( pri nt l n) val keyedRDD = rdd. keyBy( row => row( 6) . asI nst
anceOf [ I nt ] . t oDoubl e)

keyedRDD. part i t i onBy( new HashPart i t i oner( 10) ) . t ake( 10)
```

Although the hash and range partitioners are useful, they're fairly rudimentary. At times, you will need to perform some very low-level partitioning because you're working with very large data and large *key skew*. Key skew simply means that some keys have many, many more values than other keys. You want to break these keys as much as possible to improve parallelism and prevent Out Of MemoryErrors during the course of execution.

One instance might be that you need to partition more keys if and only if the key matches a certain format. For instance, we might know that there are two customers in your dataset that always crash your analysis and we need to break them up further than other customer IDs. In fact, these two are so skewed that they need to be operated on alone, whereas all of the others can be lumped into large groups. This is obviously a bit of a caricatured example, but you might see similar situations in your data, as well:

```
// in Scala import org.apache.spark.HashPartitioner class DomainPart
itioner extends Partitioner { def numPartitions = 3 def getPart
ition(key: Any): Int = { val customerId = key.asInstanceOf[Doubl
e].toInt if (customerId == 17850.0 || customerId == 12583.0)
{ return 0 } else { return new java.util.Random().nextInt
(2) + 1
 }
}
}

keyedRDD
 .partitionBy(new DomainPartitioner).map(_._1).glom().map(_.toSet.toSeq.length)
.take(5)
```

After you run this, you will see the count of results in each partition. The second two numbers will vary, because we're distributing them randomly (as you will see when we do the same in Python) but the same principles apply:

```
# in Python def partit
ionFunc(key):
 import random if key == 17850 or key
== 12583:
  return 0 else:
  return random.randint(1, 2)
keyedRDD = rdd.keyBy(lambda row: row[6]).keyedRDD\
 .partitionBy(3, partitionFunc)\
 .map(lambda x: x[0])\
 .glom()\
 .map(lambda x: len(set(x)))\
 .take(5)
```

This custom key distribution logic is available only at the RDD level. Of course, this is a simple example, but it does show the power of using arbitrary logic to distribute the data around the cluster in a physical manner.

Custom Serialization

The last advanced topic that is worth talking about is the issue of *Kryo serialization*. Any object that you hope to parallelize (or function) must be serializable:

```
// i n Scal a cl ass SomeCl ass ext ends Seri al i
zabl e {   var someVal ue = 0   def set SomeVal
ue( i : I nt ) = {    someVal ue = i    t hi s
 }
}

sc. paral l el i ze( 1 t o 10) . map( num => new SomeCl ass( ) . set SomeVal ue( num) )
```

The default serialization can be quite slow. Spark can use the Kryo library (version 2) to serialize objects more quickly. Kryo is significantly faster and more compact than Java serialization (often as much as 10x), but does not support all serializable types and requires you to register the classes you'll use in the program in advance for best performance.

You can use Kryo by initializing your job with a SparkConf and setting the value of "spark. seri al i zer" to "org. apache. spark. seri al i zer. KryoSeri al i zer" (we discuss this in the next part of the book). This setting configures the serializer used for shuffling data between worker nodes and serializing RDDs to disk. The only reason Kryo is not the default is because of the custom registration requirement, but we recommend trying it in any networkintensive application. Since Spark 2.0.0, we internally use Kryo serializer when shuffling RDDs with simple types, arrays of simple types, or string type.

Spark automatically includes Kryo serializers for the many commonly used core Scala classes covered in the AllScalaRegistrar from the Twitter chill library.

To register your own custom classes with Kryo, use the regi st erKryoCl asses method:

```
// i n Scal a
val conf = new SparkConf ( ) . set Mast er( . . . ) . set AppName( . . . ) conf . regi st erKryoCl
asses( Array( cl assOf [ MyCl ass1] , cl assOf [ MyCl ass2] ) ) val sc = new SparkCont ext (
conf )
```

Conclusion

In this chapter we discussed many of the more advanced topics regarding RDDs. Of particular note was the section on custom partitioning, which allows you very specific functions to layout your data. In Chapter 14, we discuss another of Spark's low-level tools: distributed variables.

Chapter 14. Distributed Shared Variables

In addition to the Resilient Distributed Dataset (RDD) interface, the second kind of low-level API in Spark is two types of "distributed shared variables": broadcast variables and accumulators. These are variables you can use in your user-defined functions (e.g., in a map function on an RDD or a DataFrame) that have special properties when running on a cluster. Specifically, *accumulators* let you add together data from all the tasks into a shared result (e.g., to implement a counter so you can see how many of your job's input records failed to parse), while *broadcast variables* let you save a large value on all the worker nodes and reuse it across many Spark actions without re-sending it to the cluster. This chapter discusses some of the motivation for each of these variable types as well as how to use them.

Broadcast Variables

Broadcast variables are a way you can share an immutable value efficiently around the cluster without encapsulating that variable in a function closure. The normal way to use a variable in your driver node inside your tasks is to simply reference it in your function closures (e.g., in a map operation), but this can be inefficient, especially for large variables such as a lookup table or a machine learning model. The reason for this is that when you use a variable in a closure, it must be deserialized on the worker nodes many times (one per task). Moreover, if you use the same variable in multiple Spark actions and jobs, it will be re-sent to the workers with every job instead of once.

This is where broadcast variables come in. Broadcast variables are shared, immutable variables that are cached on every machine in the cluster instead of serialized with every single task. The canonical use case is to pass around a large lookup table that fits in memory on the executors and use that in a function, as illustrated in Figure 14-1.

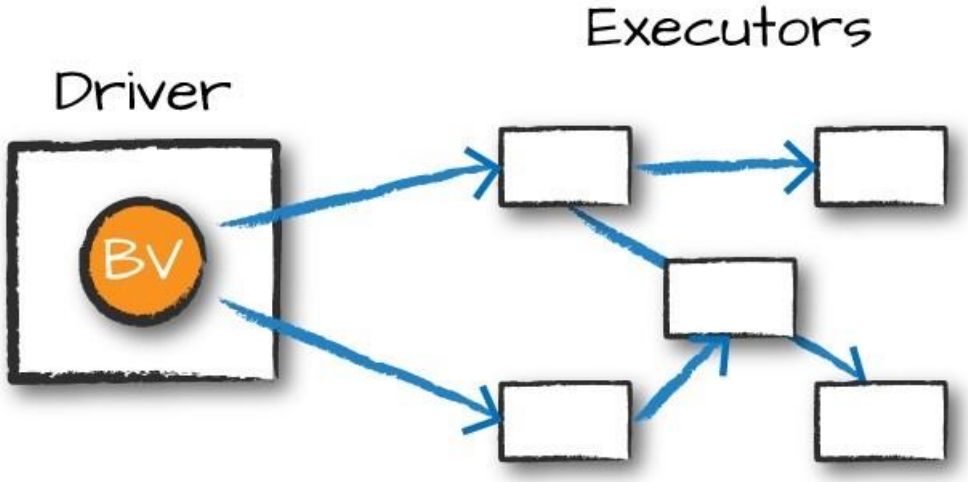


Figure 14-1. Broadcast variables

For example, suppose that you have a list of words or values:

```
// in Scala
val myCollection = "Spark The Definitive Guide : Big Data Processing Made Simple"
  .split(" ")
val words = spark.sparkContext.parallelize(myCollection, 2)

# in Python
my_collection = "Spark The Definitive Guide : Big Data Processing Made Simple"\
  .split(" ")
words = spark.sparkContext.parallelize(my_collection, 2)
```

You would like to supplement your list of words with other information that you have, which is many kilobytes, megabytes, or potentially even gigabytes in size. This is technically a right join if we thought about it in terms of SQL:

```
// in Scala
val supplementalData = Map("Spark" -> 1000, "Definitive" ->
200,                       "Big" -> -300, "Simple" -> 100)

# in Python
supplementalData = {"Spark": 1000, "Definitive": 200,
                    "Big": -300, "Simple": 100}
```

We can broadcast this structure across Spark and reference it by using suppBroadcast . This value is immutable and is lazily replicated across all nodes in the cluster when we trigger an action:

```
// in Scala
val suppBroadcast = spark.sparkContext.broadcast(supplementalData)

# in Python
suppBroadcast = spark.sparkContext.broadcast(supplementalData)
```

We reference this variable via the value method, which returns the exact value that we had earlier. This method is accessible within serialized functions without having to serialize the data. This can save you a great deal of serialization and deserialization costs because Spark transfers data more efficiently around the cluster using broadcasts:

```
// in Scala
suppBroadcast.value

# in Python
suppBroadcast.value
```

Now we could transform our RDD using this value. In this instance, we will create a key–value pair according to the value we might have in the map. If we lack the value, we will simply replace it with 0:

```
// in Scala words.map(word => (word, suppBroadcast.value.getOrElse(word,
0)))
  .sortBy(wordPair => wordPair._2)
  .collect()

# in Python words.map(lambda word: (word, suppBroadcast.value.get(word,
0))) \
  .sortBy(lambda wordPair: wordPair[1]) \
  .collect()
```

This returns the following value in Python and the same values in an array type in Scala:

```
[('Big', -300),  (
'The', 0), . . .
 ('Definitive', 200),
 ('Spark', 1000)]
```

The only difference between this and passing it into the closure is that we have done this in a much more efficient manner (Naturally, this depends on the amount of data and the number of executors. For very small data (low KBs) on small clusters, it might not be). Although this small dictionary probably is not too large of a cost, if you have a much larger value, the cost of serializing the data for every task can be quite significant.

One thing to note is that we used this in the context of an RDD; we can also use this in a UDF or in a Dataset and achieve the same result.

Accumulators

Accumulators (Figure 14-2), Spark's second type of shared variable, are a way of updating a value inside of a variety of transformations and propagating that value to the driver node in an efficient and fault-tolerant way.

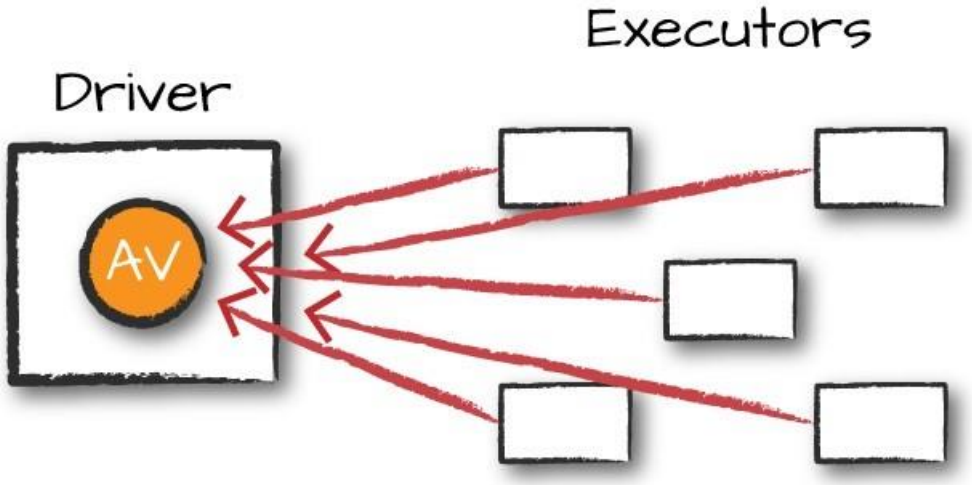


Figure 14-2. Accumulator variable

Accumulators provide a mutable variable that a Spark cluster can safely update on a per-row basis. You can use these for debugging purposes (say to track the values of a certain variable per partition in order to intelligently use it over time) or to create low-level aggregation. Accumulators are variables that are "added" to only through an associative and commutative operation and can therefore be efficiently supported in parallel. You can use them to implement counters (as in MapReduce) or sums. Spark natively supports accumulators of numeric types, and programmers can add support for new types.

For accumulator updates performed inside *actions only*, Spark guarantees that each task's update to the accumulator will be applied only once, meaning that restarted tasks will not update the value. In transformations, you should be aware that each task's update can be applied more than once if tasks or job stages are reexecuted.

Accumulators do not change the lazy evaluation model of Spark. If an accumulator is being updated within an operation on an RDD, its value is updated only once that RDD is actually computed (e.g., when you call an action on that RDD or an RDD that depends on it). Consequently, accumulator updates are not guaranteed to be executed when made within a lazy transformation like map( ) .

Accumulators can be both named and unnamed. Named accumulators will display their running results in the Spark UI, whereas unnamed ones will not.

Basic Example

Let's experiment by performing a custom aggregation on the Flight dataset that we created earlier in the book. In this example, we will use the Dataset API as opposed to the RDD API, but the extension is quite similar:

```
// i n Scal a
case cl ass Fl i ght ( DEST_COUNTRY_NAME: St ri ng,
```

```
                ORI GI N_COUNTRY_NAME: St ri ng, count : Bi gI nt )
val f l i ght s = spark. read
  . parquet ( "/dat a/f l i ght - dat a/parquet /2010- summary. parquet ")
  . as[ Fl i ght ]

# i n Pyt hon f l i ght s =
spark. read\
  . parquet ( "/dat a/f l i ght - dat a/parquet /2010- summary. parquet ")
```

Now let's create an accumulator that will count the number of flights to or from China. Even though we could do this in a fairly straightfoward manner in SQL, many things might not be so straightfoward. Accumulators provide a programmatic way of allowing for us to do these sorts of counts. The following demonstrates creating an unnamed accumulator:

```
// i n Scal a i mport org. apache. spark. ut i l . LongAccumul at
or val accUnnamed = new LongAccumul at or val acc = spark.
sparkCont ext . regi st er( accUnnamed)

# i n Pyt hon
accChi na = spark. sparkCont ext . accumul at or( 0)
```

Our use case fits a named accumulator a bit better. There are two ways to do this: a short-hand method and a long-hand one. The simplest is to use the SparkCont ext . Alternatively, we can instantiate the accumulator and register it with a name:

```
// i n Scal a val accChi na = new LongAccumul at or val accChi na2 = spark.
sparkCont ext . l ongAccumul at or( "Chi na") spark. sparkCont ext . regi st
er( accChi na, "Chi na")
```

We specify the name of the accumulator in the string value that we pass into the function, or as the second parameter into the regi st er function. Named accumulators will display in the Spark UI, whereas unnamed ones will not.

The next step is to define the way we add to our accumulator. This is a fairly straightforward function:

```
// i n Scal a def accChi naFunc( f l i ght _row: Fl i ght ) = {   val
dest i nat i on = f l i ght _row. DEST_COUNTRY_NAME   val ori
gi n = f l i ght _row. ORI GI N_COUNTRY_NAME   i f ( dest i nat
i on == "Chi na") {     accChi na. add( f l i ght _row. count . t
oLong)
  }   i f ( ori gi n == "Chi na") {     accChi na. add( f l i
ght _row. count . t oLong)
  }
}
# i n Pyt hon def accChi naFunc( f l i
ght _row) :
```

```
 dest i nat i on = f l i ght _row[ "DEST_COUNTRY_NAME"]
ori gi n = f l i ght _row[ "ORI GI N_COUNTRY_NAME"]  i f
dest i nat i on == "Chi na":
  accChi na. add( f l i ght _row[ "count "] )  i f
ori gi n == "Chi na":    accChi na. add( f l i ght
_row[ "count "] )
```

Now, let's iterate over every row in our flights dataset via the f oreach method. The reason for this is because f oreach is an action, and Spark can provide guarantees that perform only inside of actions.

The f oreach method will run once for each row in the input DataFrame (assuming that we did not filter it) and will run our function against each row, incrementing the accumulator accordingly:

```
// i n Scal a f l i ght s. f oreach( f l i ght _row => accChi naFunc( f l i ght
_row) )

# i n Pyt hon
f l i ght s. f oreach( l ambda f l i ght _row: accChi naFunc( f l i ght _row) )
```

This will complete fairly quickly, but if you navigate to the Spark UI, you can see the relevant value, on a per-Executor level, even before querying it programmatically, as demonstrated in Figure 14-3.

Summary Metrics for 1 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	0.5 s	0.5 s	0.5 s	0.5 s	0.5 s
GC Time	0 ms	0 ms	0 ms	0 ms	0 ms

Aggregated Metrics by Executor

Executor ID ▲	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tasks
driver	10.172.238.229:44026	0.5 s	1	0	1

Accumulators

Accumulable	Value
China	953

Tasks (1)

Index ▲	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Accumulators	Errors
0	210	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2017/01/17 21:33:27	0.5 s		China: 953	

Figure 14-3. Executor Spark UI

Of course, we can query it programmatically, as well. To do this, we use the val ue property:

```
// in Scala accChina.value // 953

# in Python accChina.value # 953
```

Custom Accumulators

Although Spark does provide some default accumulator types, sometimes you might want to build your own custom accumulator. In order to do this you need to subclass the AccumulatorV2 class. There are several abstract methods that you need to implement, as you can see in the example that follows. In this example, you we will add only values that are even to the accumulator. Although this is again simplistic, it should show you how easy it is to build up your own accumulators:

```
// in Scala import scala.collection.mutable.
ArrayBuffer import org.apache.spark.util.Accumul
atorV2 val arr = ArrayBuffer[BigInt]()

class EvenAccumulator extends AccumulatorV2[BigInt, BigInt] {  pri
vate var num: BigInt = 0  def reset(): Unit = {    this.num = 0
  }  def add(intValue: BigInt): Unit = {    if
(intValue % 2 == 0) {       this.num += intV
alue
   }  }  def merge(other: AccumulatorV2[BigInt, BigInt]): Unit = {
this.num += other.value
  }  def value(): BigInt = {    t
his.num
  }  def copy(): AccumulatorV2[BigInt, BigInt] = {
new EvenAccumulator
  }  def isZero(): Boolean = {    t
his.num == 0
  }
}
val acc = new EvenAccumulator val newAcc = sc.
register(acc, "evenAcc") // in Scala acc.value
// 0 flight s.foreach(flight _row => acc.add(f
light _row.count)) acc.value // 31390
```

If you are predominantly a Python user, you can also create your own custom accumulators by subclassing AccumulatorParam and using it as we saw in the previous example.

Conclusion

In this chapter, we covered distributed variables. These can be helpful tools for optimizations or for debugging. In Chapter 15, we define how Spark runs on a cluster to better understand when these can be helpful.

Part IV. Production Applications

Chapter 15. How Spark Runs on a Cluster

Thus far in the book, we focused on Spark's properties as a programming interface. We have discussed how the structured APIs take a logical operation, break it up into a logical plan, and convert that to a physical plan that actually consists of Resilient Distributed Dataset (RDD) operations that execute across the cluster of machines. This chapter focuses on what happens when Spark goes about executing that code. We discuss this in an implementation-agnostic way—this depends on neither the cluster manager that you're using nor the code that you're running. At the end of the day, all Spark code runs the same way.

This chapter covers several key topics:

- The architecture and components of a Spark Application
- The life cycle of a Spark Application inside and outside of Spark
- Important low-level execution properties, such as pipelining
- What it takes to run a Spark Application, as a segue into Chapter 16.

Let's begin with the architecture.

The Architecture of a Spark Application

In Chapter 2, we discussed some of the high-level components of a Spark Application. Let's review those again:

The Spark driver

> The driver is the process "in the driver seat" of your Spark Application. It is the controller of the execution of a Spark Application and maintains all of the state of the Spark cluster (the state and tasks of the executors). It must interface with the cluster manager in order to actually get physical resources and launch executors. At the end of the day, this is just a process on a physical machine that is responsible for maintaining the state of the application running on the cluster.

The Spark executors

> Spark executors are the processes that perform the tasks assigned by the Spark driver. Executors have one core responsibility: take the tasks assigned by the driver, run them, and report back their state (success or failure) and results. Each Spark Application has its own separate executor processes.

The cluster manager

The Spark Driver and Executors do not exist in a void, and this is where the cluster manager comes in. The cluster manager is responsible for maintaining a cluster of machines that will run your Spark Application(s). Somewhat confusingly, a cluster manager will have its own "driver" (sometimes called master) and "worker" abstractions. The core difference is that these are tied to physical machines rather than processes (as they are in Spark). Figure 15-1 shows a basic cluster setup. The machine on the left of the illustration is the *Cluster Manager Driver Node*. The circles represent daemon processes running on and managing each of the individual worker nodes. There is no Spark Application running as of yet—these are just the processes from the cluster manager.

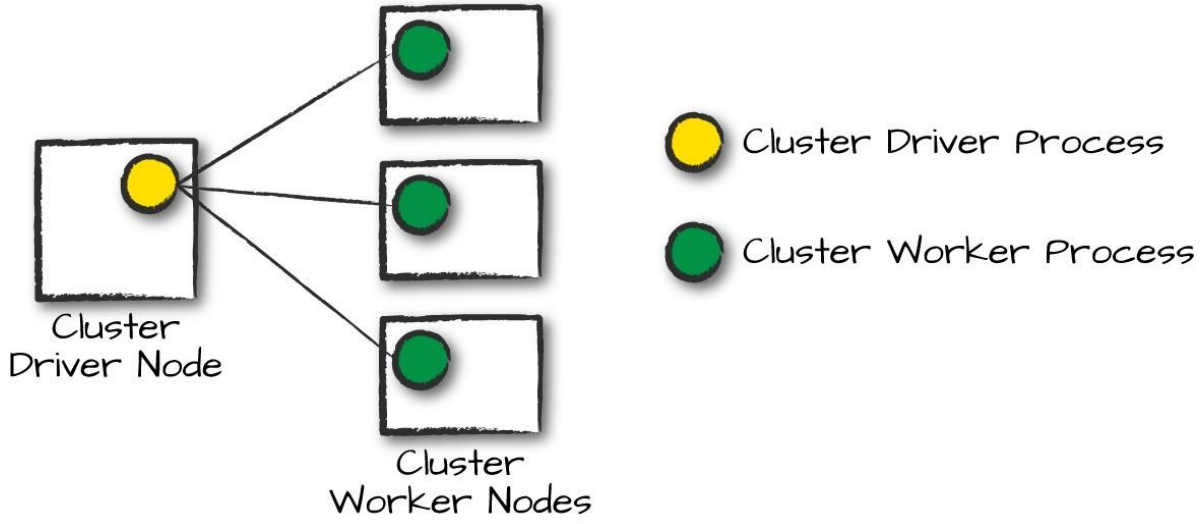


Figure 15-1. A cluster driver and worker (no Spark Application yet)

When it comes time to actually run a Spark Application, we request resources from the cluster manager to run it. Depending on how our application is configured, this can include a place to run the Spark driver or might be just resources for the executors for our Spark Application. Over the course of Spark Application execution, the cluster manager will be responsible for managing the underlying machines that our application is running on.

Spark currently supports three cluster managers: a simple built-in standalone cluster manager, Apache Mesos, and Hadoop YARN. However, this list will continue to grow, so be sure to check the documentation for your favorite cluster manager.

Now that we've covered the basic components of an application, let's walk through one of the first choices you will need to make when running your applications: choosing the execution mode.

Execution Modes

An execution *mode* gives you the power to determine where the aforementioned resources are physically located when you go to run your application. You have three modes to choose from:

- Cluster mode

- Client mode
- Local mode

We will walk through each of these in detail using Figure 15-1 as a template. In the following section, rectangles with solid borders represent Spark *driver process* whereas those with dotted borders represent the *executor processes*.

Cluster mode

Cluster mode is probably the most common way of running Spark Applications. In cluster mode, a user submits a pre-compiled JAR, Python script, or R script to a cluster manager. The cluster manager then launches the driver process on a worker node inside the cluster, in addition to the executor processes. This means that the cluster manager is responsible for maintaining all Spark Application–related processes. Figure 15-2 shows that the cluster manager placed our driver on a worker node and the executors on other worker nodes.

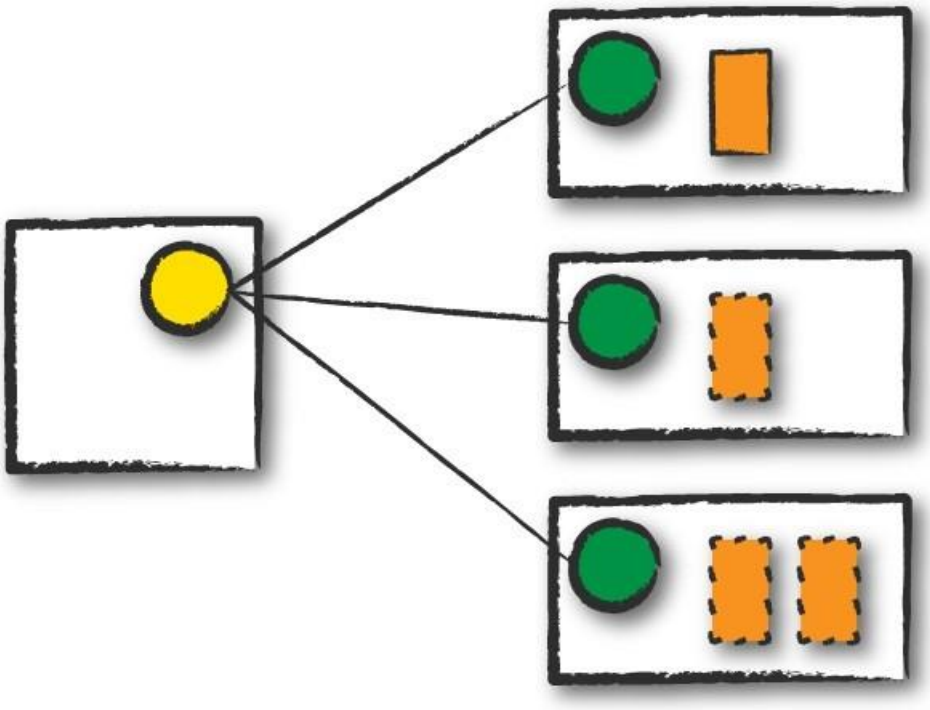


Figure 15-2. Spark's cluster mode

Client mode

Client mode is nearly the same as cluster mode except that the Spark driver remains on the client machine that submitted the application. This means that the client machine is responsible for maintaining the Spark driver process, and the cluster manager maintains the executor processses. In Figure 15-3, we are running the Spark Application from a machine that is not colocated on the cluster. These machines are commonly referred to as *gateway machines* or

edge nodes. In Figure 15-3, you can see that the driver is running on a machine outside of the cluster but that the workers are located on machines in the cluster.

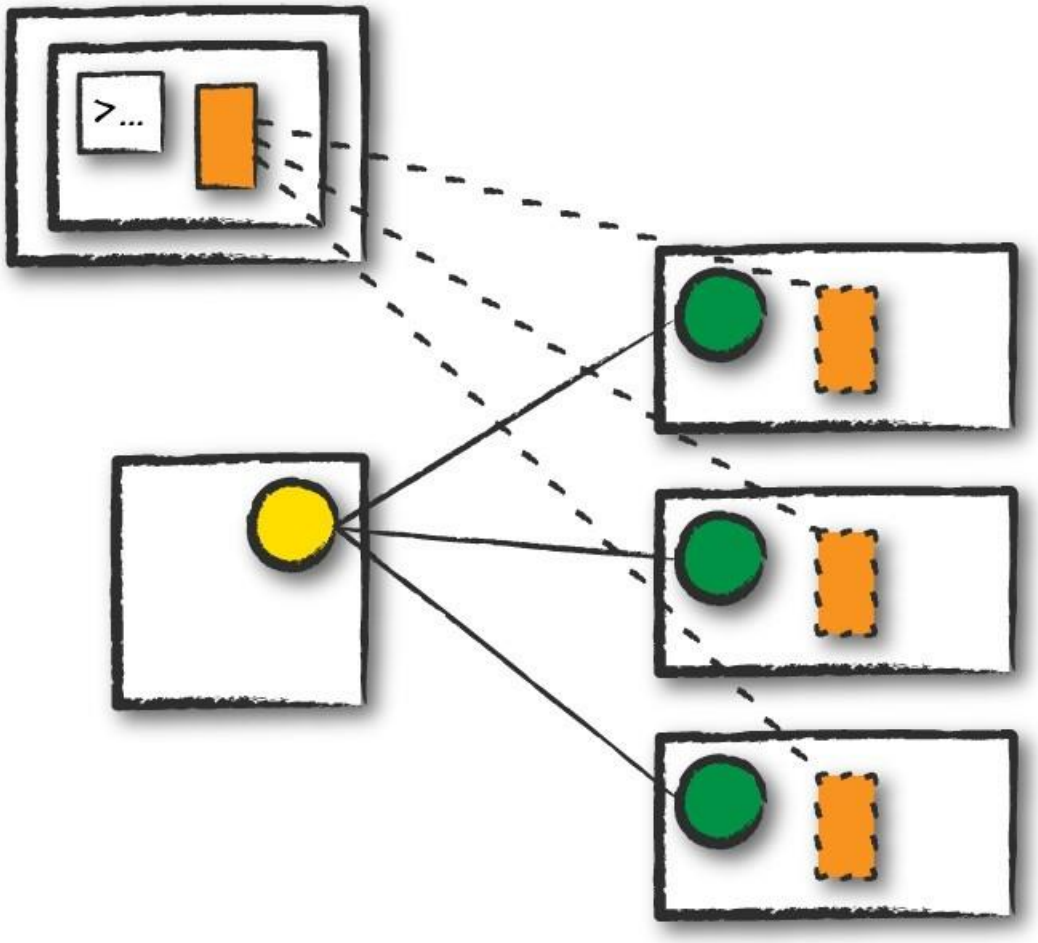


Figure 15-3. Spark's client mode

Local mode

Local mode is a significant departure from the previous two modes: it runs the entire Spark Application on a single machine. It achieves parallelism through threads on that single machine. This is a common way to learn Spark, to test your applications, or experiment iteratively with local development. However, we do not recommend using local mode for running production applications.

The Life Cycle of a Spark Application (Outside Spark)

This chapter has thus far covered the vocabulary necessary for discussing Spark Applications. It's now time to talk about the overall life cycle of Spark Applications from "outside" the actual Spark code. We will do this with an illustrated example of an application run with sparksubmi t (introduced in Chapter 3). We assume that a cluster is already running with four nodes, a driver (not a Spark driver but cluster manager driver) and three worker nodes. The actual cluster

manager does not matter at this point: this section uses the vocabulary from the previous section to walk through a step-by-step Spark Application life cycle from initialization to program exit.

> **NOTE**
>
> This section also makes use of illustrations and follows the same notation that we introduced previously. Additionally, we now introduce lines that represent network communication. Darker arrows represent communication by Spark or Spark-related processes, whereas dashed lines represent more general communication (like cluster management communication).

Client Request

The first step is for you to submit an actual application. This will be a pre-compiled JAR or library. At this point, you are executing code on your local machine and you're going to make a request to the cluster manager driver node (Figure 15-4). Here, we are explicitly asking for resources for the *Spark driver process* only. We assume that the cluster manager accepts this offer and places the driver onto a node in the cluster. The client process that submitted the original job exits and the application is off and running on the cluster.

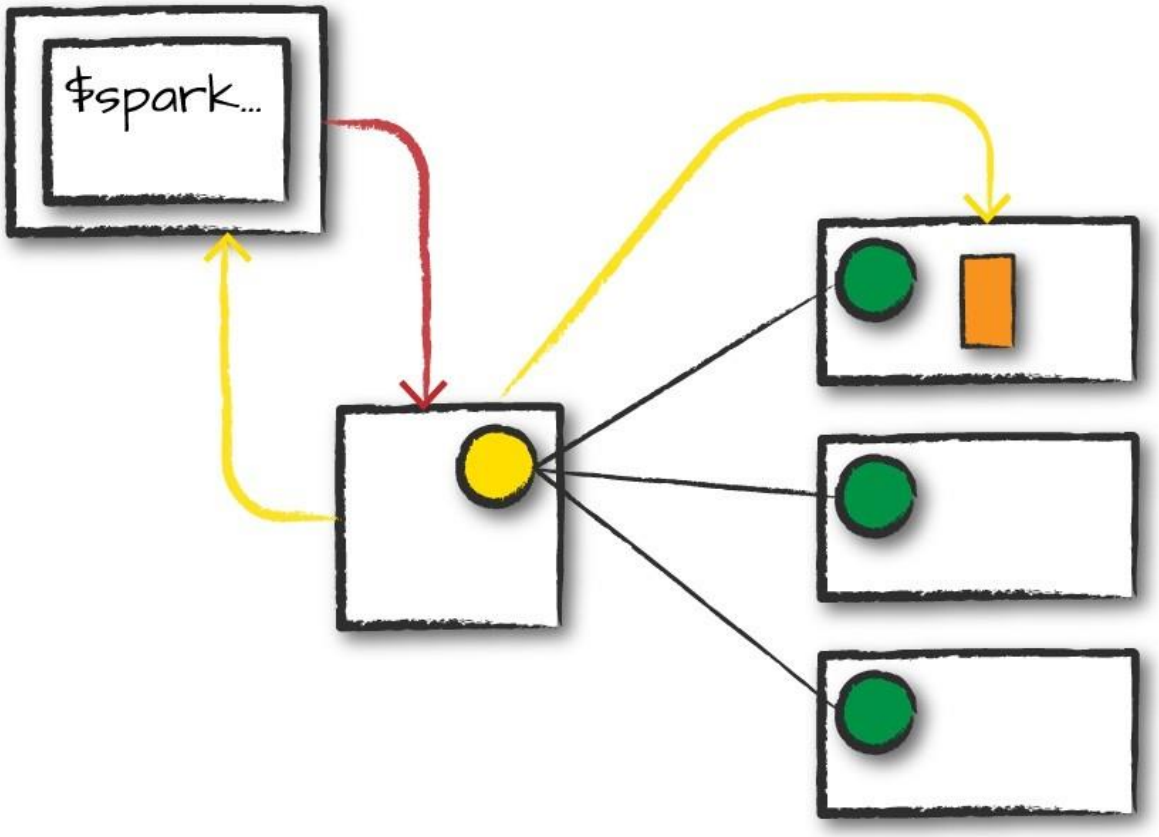


Figure 15-4. Requesting resources for a driver

To do this, you'll run something like the following command in your terminal:

```
. /bi n/spark- submi t \
- - cl ass <mai n- cl ass> \
- - mast er <mast er- url > \
- - depl oy- mode cl ust er \
- - conf <key>=<val ue> \
  . . . # ot her opt i ons    <appl i
cat i on- j ar> \
  [ appl i cat i on- argument s]
```

Launch

Now that the driver process has been placed on the cluster, it begins running user code (Figure 15-5). This code must include a SparkSessi on that initializes a Spark cluster (e.g., driver + executors). The SparkSession will subsequently communicate with the cluster manager (the darker line), asking it to launch Spark executor processes across the cluster (the lighter lines). The number of executors and their relevant configurations are set by the user via the command-line arguments in the original spark- submi t call.

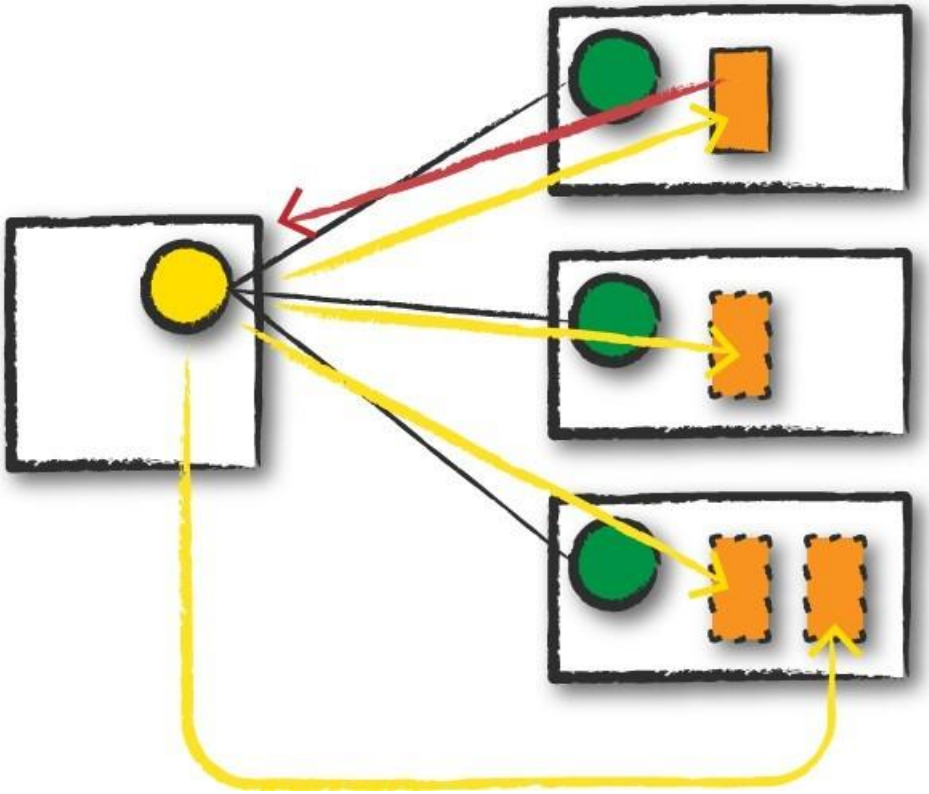


Figure 15-5. Launching the Spark Application

The cluster manager responds by launching the executor processes (assuming all goes well) and sends the relevant information about their locations to the driver process. After everything is hooked up correctly, we have a "Spark Cluster" as you likely think of it today.

Execution

Now that we have a "Spark Cluster," Spark goes about its merry way executing code, as shown in Figure 15-6. The driver and the workers communicate among themselves, executing code and moving data around. The driver schedules tasks onto each worker, and each worker responds with the status of those tasks and success or failure. (We cover these details shortly.)

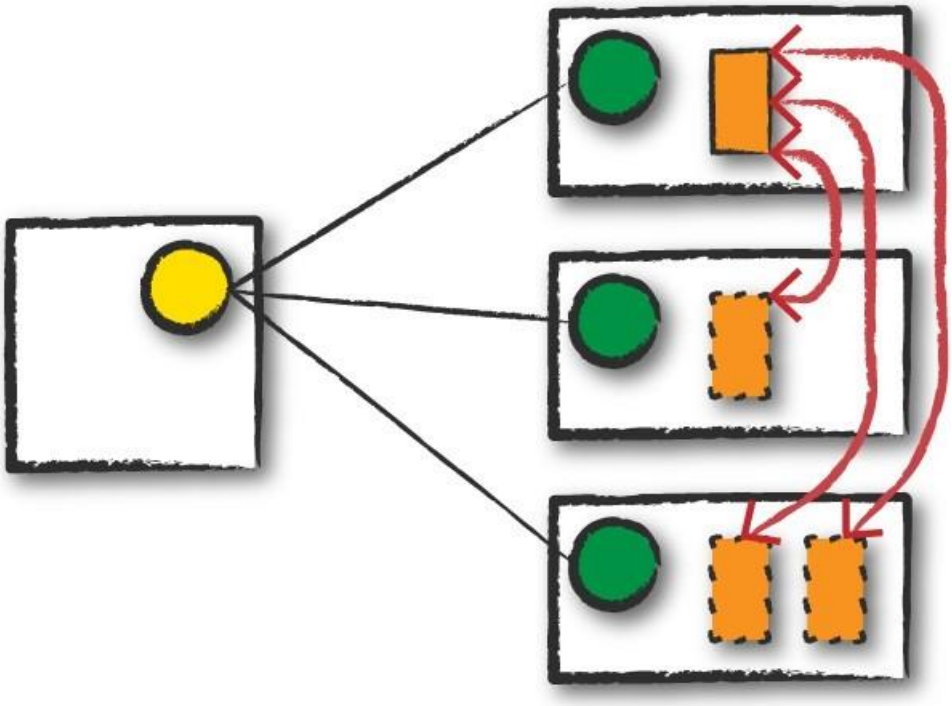


Figure 15-6. Application execution

Completion

After a Spark Application completes, the driver processs exits with either success or failure (Figure 15-7). The cluster manager then shuts down the executors in that Spark cluster for the driver. At this point, you can see the success or failure of the Spark Application by asking the cluster manager for this information.

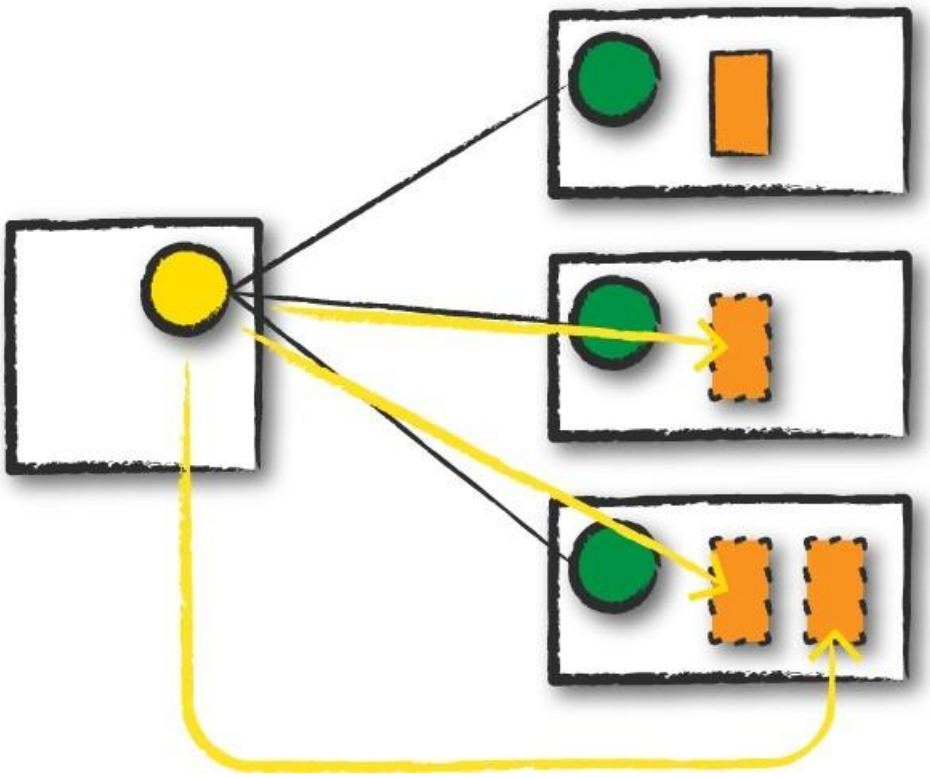


Figure 15-7. Shutting down the application

The Life Cycle of a Spark Application (Inside Spark)

We just examined the life cycle of a Spark Application outside of user code (basically the infrastructure that supports Spark), but it's arguably more important to talk about what happens within Spark when you run an application. This is "user-code" (the actual code that you write that defines your Spark Application). Each application is made up of one or more *Spark jobs*. Spark jobs within an application are executed serially (unless you use threading to launch multiple actions in parallel).

The SparkSession

The first step of any Spark Application is creating a SparkSession. In many interactive modes, this is done for you, but in an application, you must do it manually.

Some of your legacy code might use the new SparkCont ext pattern. This should be avoided in favor of the builder method on the SparkSessi on, which more robustly instantiates the Spark and SQL Contexts and ensures that there is no context conflict, given that there might be multiple libraries trying to create a session in the same Spark Appication:

```
// Creat i ng a SparkSessi on i n Scal a i mport org. apache. spark. sql . SparkSessi on val
spark = SparkSessi on. bui l der( ) . appName( "Dat abri cks Spark Exampl e")   . conf i g(
"spark. sql . warehouse. di r", "/user/hi ve/warehouse")
```

```
  . get OrCreat e( )

# Creat i ng a SparkSessi on i n Pyt hon f rom pyspark. sql i mport SparkSessi on spark
= SparkSessi on. bui l der. mast er( "l ocal ") . appName( "Word Count ") \
  . conf i g( "spark. some. conf i g. opt i on", "some- val ue") \
  . get OrCreat e( )
```

After you have a SparkSessi on, you should be able to run your Spark code. From the SparkSessi on, you can access all of low-level and legacy contexts and configurations accordingly, as well. Note that the SparkSessi on class was only added in Spark 2.X. Older code you might find would instead directly create a SparkCont ext and a SQLCont ext for the structured APIs.

The SparkContext

A SparkCont ext object within the SparkSessi on represents the connection to the Spark cluster. This class is how you communicate with some of Spark's lower-level APIs, such as RDDs. It is commonly stored as the variable sc in older examples and documentation. Through a SparkCont ext , you can create RDDs, accumulators, and broadcast variables, and you can run code on the cluster.

For the most part, you should not need to explicitly initialize a SparkCont ext ; you should just be able to access it through the SparkSessi on. If you do want to, you should create it in the most general way, through the get OrCreat e method:

```
// i n Scal a i mport org. apache. spark.
SparkCont ext val sc = SparkCont ext . get
OrCreat e( )
```

THE SPARKSESSION, SQLCONTEXT, AND HIVECONTEXT

In previous versions of Spark, the SQLCont ext and Hi veCont ext provided the ability to work with DataFrames and Spark SQL and were commonly stored as the variable sql Cont ext in examples, documentation, and legacy code. As a historical point, Spark 1.X had effectively two contexts. The SparkCont ext and the SQLCont ext . These two each performed different things. The former focused on more fine-grained control of Spark's central abstractions, whereas the latter focused on the higher-level tools like Spark SQL. In Spark 2.X, the communtiy combined the two APIs into the centralized SparkSessi on that we have today. However, both of these APIs still exist and you can access them via the SparkSessi on. It is important to note that you should never need to use the SQLCont ext and rarely need to use the SparkCont ext .

After you initialize your SparkSession, it's time to execute some code. As we know from previous chapters, all Spark code compiles down to RDDs. Therefore, in the next section, we will take some logical instructions (a DataFrame job) and walk through, step by step, what happens over time.

Logical Instructions

As you saw in the beginning of the book, Spark code essentially consists of transformations and actions. How you build these is up to you—whether it's through SQL, low-level RDD manipulation, or machine learning algorithms. Understanding how we take declarative instructions like DataFrames and convert them into physical execution plans is an important step to understanding how Spark runs on a cluster. In this section, be sure to run this in a fresh environment (a new Spark shell) to follow along with the job, stage, and task numbers.

Logical instructions to physical execution

We mentioned this in Part II, but it's worth reiterating so that you can better understand how Spark takes your code and actually runs the commands on the cluster. We will walk through some more code, line by line, explain what's happening behind the scenes so that you can walk away with a better understanding of your Spark Applications. In later chapters, when we discuss monitoring, we will perform a more detailed tracking of a Spark job through the Spark UI. In this current example, we'll take a simpler approach. We are going to do a three-step job: using a simple DataFrame, we'll repartition it, perform a value-by-value manipulation, and then aggregate some values and collect the final result.

> **NOTE**
>
> This code was written and runs with Spark 2.2 in Python (you'll get the same result in Scala, so we've omitted it). The number of jobs is unlikely to change drastically but there might be improvements to Spark's underlying optimizations that change physical execution strategies.

```
# in Python
df1 = spark.range(2, 10000000, 2)
df2 = spark.range(2, 10000000, 4)
step1 = df1.repartition(5)
step12 = df2.repartition(6)
step2 = step1.selectExpr("id * 5 as id")
step3 = step2.join(step12, ["id"])
step4 = step3.selectExpr("sum(id)")
step4.collect() # 2500000000000
```

When you run this code, we can see that your action triggers one complete Spark job. Let's take a look at the explain plan to ground our understanding of the physical execution plan. We can access this information on the SQL tab (after we actually run a query) in the Spark UI, as well: `step4.explain()`

```
== Physical Plan ==
*HashAggregate(keys=[], functions=[sum(id#15L)])
+- Exchange SinglePartition
   +- *HashAggregate(keys=[], functions=[partial_sum(id#15L)])
```

```
+- *Proj ect [ i d#15L]
  +- *Sort MergeJ oi n [ i d#15L] , [ i d#10L] , I nner
    : - *Sort [ i d#15L ASC NULLS FI RST] , f al se, 0
    :  +- Exchange hashpart i t i oni ng( i d#15L, 200)
    :     +- *Proj ect [ ( i d#7L * 5) AS i d#15L]
    :        +- Exchange RoundRobi nPart i t i oni ng( 5)
    :           +- *Range ( 2, 10000000, st ep=2, spl i t s=8)
    +- *Sort [ i d#10L ASC NULLS FI RST] , f al se, 0
      +- Exchange hashpart i t i oni ng( i d#10L, 200)
        +- Exchange RoundRobi nPart i t i oni ng( 6)
          +- *Range ( 2, 10000000, st ep=4, spl i t s=8)
```

What you have when you call col l ect (or any action) is the execution of a Spark *job* that individually consist of *stages* and *tasks*. Go to localhost:4040 if you are running this on your local machine to see the Spark UI. We will follow along on the "jobs" tab eventually jumping to stages and tasks as we proceed to further levels of detail.

A Spark Job

In general, there should be one Spark job for one action. Actions always return results. Each job breaks down into a series of *stages*, the number of which depends on how many shuffle operations need to take place.

This job breaks down into the following stages and tasks:

- Stage 1 with 8 Tasks
- Stage 2 with 8 Tasks
- Stage 3 with 6 Tasks
- Stage 4 with 5 Tasks
- Stage 5 with 200 Tasks
- Stage 6 with 1 Task

I hope you're at least somewhat confused about how we got to these numbers so that we can take the time to better understand what is going on!

Stages

Stages in Spark represent groups of tasks that can be executed together to compute the same operation on multiple machines. In general, Spark will try to pack as much work as possible (i.e., as many transformations as possible inside your job) into the same stage, but the engine starts new stages after operations called *shuffles*. A shuffle represents a physical repartitioning of the data—for example, sorting a DataFrame, or grouping data that was loaded from a file by key (which requires sending records with the same key to the same node). This type of

repartitioning requires coordinating across executors to move data around. Spark starts a new stage after each shuffle, and keeps track of what order the stages must run in to compute the final result.

In the job we looked at earlier, the first two stages correspond to the range that you perform in order to create your DataFrames. By default when you create a DataFrame with range, it has eight partitions. The next step is the repartitioning. This changes the number of partitions by shuffling the data. These DataFrames are shuffled into six partitions and five partitions, corresponding to the number of tasks in stages 3 and 4.

Stages 3 and 4 perform on each of those DataFrames and the end of the stage represents the join (a shuffle). Suddenly, we have 200 tasks. This is because of a Spark SQL configuration. The spark. sql . shuf f l e. part i t i ons default value is 200, which means that when there is a shuffle performed during execution, it outputs 200 shuffle partitions by default. You can change this value, and the number of output partitions will change.

> **TIP**
>
> We cover the number of partitions in a bit more detail in Chapter 19 because it's such an important parameter. This value should be set according to the number of cores in your cluster to ensure efficient execution. Here's how to set it:
>
> ```
> spark. conf . set ( "spark. sql . shuf f l e. part i t i ons", 50)
> ```

A good rule of thumb is that the number of partitions should be larger than the number of executors on your cluster, potentially by multiple factors depending on the workload. If you are running code on your local machine, it would behoove you to set this value lower because your local machine is unlikely to be able to execute that number of tasks in parallel. This is more of a default for a cluster in which there might be many more executor cores to use. Regardless of the number of partitions, that entire stage is computed in parallel. The final result aggregates those partitions individually, brings them all to a single partition before finally sending the final result to the driver. We'll see this configuration several times over the course of this part of the book.

Tasks

Stages in Spark consist of *tasks*. Each task corresponds to a combination of blocks of data and a set of transformations that will run on a single executor. If there is one big partition in our dataset, we will have one task. If there are 1,000 little partitions, we will have 1,000 tasks that can be executed in parallel. A task is just a unit of computation applied to a unit of data (the partition). Partitioning your data into a greater number of partitions means that more can be executed in parallel. This is not a panacea, but it is a simple place to begin with optimization.

Execution Details

Tasks and stages in Spark have some important properties that are worth reviewing before we close out this chapter. First, Spark automatically *pipelines* stages and tasks that can be done together, such as a map operation followed by another map operation. Second, for all shuffle operations, Spark writes the data to stable storage (e.g., disk), and can reuse it across multiple jobs. We'll discuss these concepts in turn because they will come up when you start inspecting applications through the Spark UI.

Pipelining

An important part of what makes Spark an "in-memory computation tool" is that unlike the tools that came before it (e.g., MapReduce), Spark performs as many steps as it can at one point in time before writing data to memory or disk. One of the key optimizations that Spark performs is *pipelining*, which occurs at and below the RDD level. With pipelining, any sequence of operations that feed data directly into each other, without needing to move it across nodes, is collapsed into a single stage of tasks that do all the operations together. For example, if you write an RDD-based program that does a map, then a f i l t er, then another map, these will result in a *single* stage of tasks that immediately read each input record, pass it through the first map, pass it through the f i l t er, and pass it through the last map function if needed. This pipelined version of the computation is much faster than writing the intermediate results to memory or disk after each step. The same kind of pipelining happens for a DataFrame or SQL computation that does a sel ect , f i l t er, and sel ect .

From a practical point of view, pipelining will be transparent to you as you write an application —the Spark runtime will automatically do it—but you will see it if you ever inspect your application through the Spark UI or through its log files, where you will see that multiple RDD or DataFrame operations were pipelined into a single stage.

Shuffle Persistence

The second property you'll sometimes see is shuffle persistence. When Spark needs to run an operation that has to move data *across* nodes, such as a reduce-by-key operation (where input data for each key needs to first be brought together from many nodes), the engine can't perform pipelining anymore, and instead it performs a cross-network shuffle. Spark always executes shuffles by first having the "source" tasks (those sending data) write *shuffle files* to their local disks during their execution stage. Then, the stage that does the grouping and reduction launches and runs tasks that fetch their corresponding records from each shuffle file and performs that computation (e.g., fetches and processes the data for a specific range of keys). Saving the shuffle files to disk lets Spark run this stage later in time than the source stage (e.g., if there are not enough executors to run both at the same time), and also lets the engine re-launch reduce tasks on failure without rerunning all the input tasks.

One side effect you'll see for shuffle persistence is that running a new job over data that's already been shuffled does not rerun the "source" side of the shuffle. Because the shuffle files were already written to disk earlier, Spark knows that it can use them to run the later stages of the job, and it need not redo the earlier ones. In the Spark UI and logs, you will see the preshuffle stages marked as "skipped". This automatic optimization can save time in a workload that runs multiple jobs over the same data, but of course, for even better performance you can perform your own caching with the DataFrame or RDD cache method, which lets you control exactly which data is saved and where. You'll quickly grow accustomed to this behavior after you run some Spark actions on aggregated data and inspect them in the UI.

Conclusion

In this chapter, we discussed what happens to Spark Applications when we go to execute them on a cluster. This means how the cluster will actually go about running that code as well as what happens within Spark Applications during the process. At this point, you should feel quite comfortable understanding what happens within and outside of a Spark Application. This will give you a starting point for debugging your applications. Chapter 16 will discuss writing Spark Applications and the things you should consider when doing so.

Chapter 16. Developing Spark Applications

In Chapter 15, you learned about how Spark runs your code on the cluster. We'll now show you how easy it is to develop a standalone Spark application and deploy it on a cluster. We'll do this using a simple template that shares some easy tips for how to structure your applications, including setting up build tools and unit testing. This template is available in the book's code repository. This template is not really necessary, because writing applications from scratch isn't hard, but it helps. Let's get started with our first application.

Writing Spark Applications

Spark Applications are the combination of two things: a Spark cluster and your code. In this case, the cluster will be local mode and the application will be one that is pre-defined. Let's walk through an application in each langauge.

A Simple Scala-Based App

Scala is Spark's "native" language and naturally makes for a great way to write applications. It's really no different than writing a Scala application.

TIP

Scala can seem intimidating, depending on your background, but it's worth learning if only to understand Spark just a bit better. Additionally, you do not need to learn all the language's ins and outs; begin with the basics and you'll see that it's easy to be productive in Scala in no time. Using Scala will also open up a lot of doors. With a little practice, it's not to difficult to do code-level tracing through Spark's codebase.

You can build applications using sbt or Apache Maven, two Java Virtual Machine (JVM)–based build tools. As with any build tool, they each have their own quirks, but it's probably easiest to begin with sbt. You can download, install, and learn about sbt on the sbt website. You can install Maven from its respective website, as well.

To configure an sbt build for our Scala application, we specify a *build.sbt* file to manage the package information. Inside the *build.sbt* file, there are a few key things to include:

- Project metadata (package name, package versioning information, etc.)
- Where to resolve dependencies
- Dependencies needed for your library

There are many more options that you can specify; however, they are beyond the scope of this book (you can find information about this on the web and in the sbt documentation). There are also some books on the subject that can serve as a helpful reference as soon as you've gone beyond anything nontrivial. Here's what a sample Scala *built.sbt* file might look like (and the one that we include in the template). Notice how we must specify the Scala version as well as the Spark version:

```
name : = "exampl e" organi zat i on : =
"com. dat abri cks" versi on : = "0. 1-
SNAPSHOT" scal aVersi on : = "2. 11. 8"

// Spark I nf ormat i on val
sparkVersi on = "2. 2. 0"

// al l ows us t o i ncl ude spark packages resol vers += "bi nt
ray- spark- packages" at "ht t ps: //dl . bi nt ray. com/spark-
packages/maven/"

resol vers += "Typesaf e Si mpl e Reposi t ory" at
 "ht t p: //repo. t ypesaf e. com/t ypesaf e/si mpl e/maven- rel eases/"

resol vers += "MavenReposi t ory" at "ht t
ps: //mvnreposi t ory. com/"

l i braryDependenci es ++= Seq(
 // spark core
 "org. apache. spark" %% "spark- core" % sparkVersi on,
```

```
  "org. apache. spark" %% "spark- sql " % sparkVersi on,
// t he rest of t he f i l e i s omi t t ed f or brevi t y
)
```

Now that we've defined the build file, we can actually go about adding code to our project. We'll use the standard Scala project structure, which you can find in the sbt reference manual (this is the same directory structure as Maven projects):

```
src/   mai n/
resources/
      <f i l es t o i ncl ude i n mai n j ar here>     scal a/
      <mai n Scal a sources>     j ava/
      <mai n J ava sources>   t est /
resources
      <f i l es t o i ncl ude i n t est j ar here>     scal a/
      <t est Scal a sources>
    j ava/
      <t est J ava sources>
```

We put the source code in the Scala and Java directories. In this case, we put something like the following in a file; this initializes the SparkSessi on, runs the application, and then exits:

```
obj ect Dat aFrameExampl e ext ends Seri al i zabl e {   def
mai n( args: Array[ St ri ng] ) = {     val pat hToDat aFol der
= args( 0)

   // st art up t he SparkSessi on
   // al ong wi t h expl i ci t l y set t i ng a gi ven conf i g     val spark = SparkSessi
on. bui l der( ) . appName( "Spark Exampl e")
     . conf i g( "spark. sql . warehouse. di r", "/user/hi ve/warehouse")
     . get OrCreat e( )

   // udf regi st rat i on     spark. udf . regi st er( "myUDF", someUDF( _: St ri ng) : St ri ng)
val df = spark. read. j son( pat hToDat aFol der + "dat a. j son")     val mani pul at ed = df .
groupBy( expr( "myUDF( group) ") ) . sum( ) . col l ect ( )      . f oreach( x => pri nt l n( x) )

 }
}
```

Notice how we defined a mai n class that we can run from the command line when we use spark- submi t to submit it to our cluster for execution.

Now that we have our project set up and have added some code to it, it's time to build it. We can use sbt assembl e to build an "uber-jar" or "fat-jar" that contains all of the dependencies in one JAR. This can be simple for some deployments but cause complications (especially dependency conflicts) for others. A lighter-weight approach is to run sbt package, which will

gather all of your dependencies into the target folder but will not package all of them into one big JAR.

Running the application

The target folder contains the JAR that we can use as an argument to spark- submi t . After building the Scala package, you end up with something that you can spark- submi t on your local machine by using the following code (this snippet takes advantage of aliasing to create the $SPARK_HOME variable; you could replace $SPARK_HOME with the exact directory that contains your downloaded version of Spark):

```
$SPARK_HOME/bi n/spark- submi t \
- - cl ass com. dat abri cks. exampl e. Dat aFrameExampl e \
- - mast er l ocal \   t arget /scal a- 2. 11/exampl e_2. 11- 0. 1- SNAPSHOT. j ar "hel l o"
```

Writing Python Applications

Writing PySpark Applications is really no different than writing normal Python applications or packages. It's quite similar to writing command-line applications in particular. Spark doesn't have a build concept, just Python scripts, so to run an application, you simply execute the script against the cluster.

To facilitate code reuse, it is common to package multiple Python files into egg or ZIP files of Spark code. To include those files, you can use the - - py- f i l es argument of spark- submi t to add *.py*, *.zip*, or *.egg* files to be distributed with your application.

When it's time to run your code, you create the equivalent of a "Scala/Java main class" in Python. Specify a certain script as an executable script that builds the SparkSessi on. This is the one that we will pass as the main argument to spark- submi t :

```
# i n Pyt hon f rom __f ut ure__ i mport pri nt _f
unct i on i f __name__ == ' __mai n__' :
   f rom pyspark. sql i mport SparkSessi on    spark
= SparkSessi on. bui l der \       . mast er( "l ocal ")
\
      . appName( "Word Count ") \
      . conf i g( "spark. some. conf i g. opt i on", "some- val ue") \
      . get OrCreat e( )    pri nt ( spark. range( 5000) . where( "i d > 500") . sel ect Expr( "sum( i d) ")

. col l ect ( ) )
```

When you do this, you're going to get a SparkSessi on that you can pass around your application. It is best practice to pass around this variable at runtime rather than instantiating it within every Python class.

One helpful tip when developing in Python is to use pi p to specify PySpark as a dependency. You can do this by running the command pi p i nst al l pyspark. This allows you to use it in a way that you might use other Python packages. This makes for very helpful code completion in many

edi t ors, as well. This is brand new in Spark 2.2, so it might take a version or two to be completely production ready, but Python is very popular in the Spark community, and it's sure to be a cornerstone of Spark's future.

Running the application

After you've written your code, it's time to submit it for execution. (We're executing the same code that we have in the project template.) You just need to call spark- submi t with that information:

```
$SPARK_HOME/bi n/spark- submi t - - mast er l ocal pyspark_t empl at e/mai n. py
```

Writing Java Applications

Writing Java Spark Applications is, if you squint, the same as writing Scala applications. The core differences involve how you specify your dependencies.

This example assumes that you are using Maven to specify your dependencies. In this case, you'll use the following format. In Maven, you must add the Spark Packages repository so that you can fetch dependencies from those locations:

```
<dependenci es>
  <dependency>
    <groupI d>org. apache. spark</groupI d>
    <art i f act I d>spark- core_2. 11</art i f act I d>
    <versi on>2. 1. 0</versi on>
  </dependency>
  <dependency>
    <groupI d>org. apache. spark</groupI d>
    <art i f act I d>spark- sql _2. 11</art i f act I d>
    <versi on>2. 1. 0</versi on>
  </dependency>
  <dependency>
    <groupI d>graphf rames</groupI d>
    <art i f act I d>graphf rames</art i f act I d>
    <versi on>0. 4. 0- spark2. 1- s_2. 11</versi on>
  </dependency>
</dependenci es>
<reposi t ori es>
  <! - - l i st of ot her reposi t ori es - - >
  <reposi t ory>
    <i d>SparkPackagesRepo</i d>
    <url >ht t p: //dl . bi nt ray. com/spark- packages/maven</url >
  </reposi t ory>
</reposi t ori es>
```

Naturally, you follow the same directory structure as in the Scala project version (seeing as they both conform to the Maven specification). We then just follow the relevant Java examples to actually build and execute the code. Now we can create a simple example that specifies a mai n class for us to execute against (more on this at the end of the chapter):

```
i mport org. apache. spark. sql . SparkSessi on; publ i c cl
ass Si mpl eExampl e {
   publ i c st at i c voi d mai n( St ri ng[ ] args) {
      SparkSessi on spark = SparkSessi on
            . bui l der( )              . get OrCreat e( )
;       spark. range( 1, 2000) . count ( ) ;
   }
}
```

We then package it by using mvn package (you need to have Maven installed to do so).

Running the application

This operation is going to be the exact same as running the Scala application (or the Python application, for that matter). Simply use spark- submi t :

```
$SPARK_HOME/bi n/spark- submi t \
- - cl ass com. dat abri cks. exampl e. Si mpl eExampl e \
- - mast er l ocal \
  t arget /spark- exampl e- 0. 1- SNAPSHOT. j ar "hel l o"
```

Testing Spark Applications

You now know what it takes to write and run a Spark Application, so let's move on to a less exciting but still very important topic: testing. Testing Spark Applications relies on a couple of key principles and tactics that you should keep in mind as you're writing your applications.

Strategic Principles

Testing your data pipelines and Spark Applications is just as important as actually writing them. This is because you want to ensure that they are resilient to future change, in data, logic, and output. In this section, we'll first discuss *what* you might want to test in a typical Spark Application, then discuss *how* to organize your code for easy testing.

Input data resilience

Being resilient to different kinds of input data is something that is quite fundamental to how you write your data pipelines. The data will change because the business needs will change. Therefore your Spark Applications and pipelines should be resilient to at least some degree of change in the input data or otherwise ensure that these failures are handled in a graceful and resilient way. For the most part this means being smart about writing your tests to handle those

edge cases of different inputs and making sure that the pager only goes off when it's something that is truly important.

Business logic resilience and evolution

The business logic in your pipelines will likely change as well as the input data. Even more importantly, you want to be sure that what you're deducing from the raw data is what you actually think that you're deducing. This means that you'll need to do robust logical testing with realistic data to ensure that you're actually getting what you want out of it. One thing to be wary of here is trying to write a bunch of "Spark Unit Tests" that just test Spark's functionality. You don't want to be doing that; instead, you want to be testing your business logic and ensuring that the complex business pipeline that you set up is actually doing what you think it should be doing.

Resilience in output and atomicity

Assuming that you're prepared for departures in the structure of input data and that your business logic is well tested, you now want to ensure that your output structure is what you expect. This means you will need to gracefully handle output schema resolution. It's not often that data is simply dumped in some location, never to be read again—most of your Spark pipelines are probably feeding other Spark pipelines. For this reason you're going to want to make certain that your downstream consumers understand the "state" of the data—this could mean how frequently it's updated as well as whether the data is "complete" (e.g., there is no late data) or that there won't be any last-minute corrections to the data.

All of the aforementioned issues are principles that you should be thinking about as you build your data pipelines (actually, regardless of whether you're using Spark). This strategic thinking is important for laying down the foundation for the system that you would like to build.

Tactical Takeaways

Although strategic thinking is important, let's talk a bit more in detail about some of the tactics that you can actually use to make your application easy to test. The highest value approach is to verify that your business logic is correct by employing proper unit testing and to ensure that you're resilient to changing input data or have structured it so that schema evolution will not become unwielding in the future. The decision for how to do this largely falls on you as the developer because it will vary according to your business domain and domain expertise.

Managing SparkSessions

Testing your Spark code using a unit test framework like JUnit or ScalaTest is relatively easy because of Spark's local mode—just create a local mode SparkSession as part of your test harness to run it. However, to make this work well, you should try to perform dependency injection as much as possible when managing SparkSessions in your code. That is, initialize the

SparkSessi on only once and pass it around to relevant functions and classes at runtime in a way that makes it easy to substitute during testing. This makes it much easier to test each individual function with a dummy SparkSession in unit tests.

Which Spark API to Use?

Spark offers several choices of APIs, ranging from SQL to DataFrames and Datasets, and each of these can have different impacts for maintainability and testability of your application. To be perfectly honest, the right API depends on your team and its needs: some teams and projects will need the less strict SQL and DataFrame APIs for speed of development, while others will want to use type-safe Datasets or RDDs.

In general, we recommend documenting and testing the input and output types of each function regardless of which API you use. The type-safe API automatically enforces a minimal contract for your function that makes it easy for other code to build on it. If your team prefers to use DataFrames or SQL, then spend some time to document *and test* what each function returns and what types of inputs it accepts to avoid surprises later, as in any dynamically typed programming language. While the lower-level RDD API is also statically typed, we recommend going into it only if you need low-level features such as partitioning that are not present in Datasets, which should not be very common; the Dataset API allows more performance optimizations and is likely to provide even more of them in the future.

A similar set of considerations applies to which programming language to use for your application: there certainly is no right answer for every team, but depending on your needs, each language will provide different benefits. We generally recommend using statically typed languages like Scala and Java for larger applications or those where you want to be able to drop into low-level code to fully control performance, but Python and R may be significantly better in other cases—for example, if you need to use some of their other libraries. Spark code should easily be testable in the standard unit testing frameworks in every language.

Connecting to Unit Testing Frameworks

To unit test your code, we recommend using the standard frameworks in your langage (e.g., JUnit or ScalaTest), and setting up your test harnesses to create and clean up a SparkSession for each test. Different frameworks offer different mechanisms to do this, such as “before” and “after” methods. We have included some sample unit testing code in the application templates for this chapter.

Connecting to Data Sources

As much as possible, you should make sure your testing code does not connect to production data sources, so that developers can easily run it in isolation if these data sources change. One easy way to make this happen is to have all your business logic functions take DataFrames or Datasets as input instead of directly connecting to various sources; after all, subsequent code

will work the same way no matter what the data source was. If you are using the structured APIs in Spark, another way to make this happen is named tables: you can simply register some dummy datasets (e.g., loaded from small text file or from in-memory objects) as various table names and go from there.

The Development Process

The development process with Spark Applications is similar to development workflows that you have probably already used. First, you might maintain a scratch space, such as an interactive notebook or some equivalent thereof, and then as you build key components and algorithms, you move them to a more permanent location like a library or package. The notebook experience is one that we often recommend (and are using to write this book) because of its simplicity in experimentation. There are also some tools, such as Databricks, that allow you to run notebooks as production applications as well.

When running on your local machine, the spark- shel l and its various language-specific implementations are probably the best way to develop applications. For the most part, the shell is for interactive applications, whereas spark- submi t is for production applications on your Spark cluster. You can use the shell to interactively run Spark, just as we showed you at the beginning of this book. This is the mode with which you will run PySpark, Spark SQL, and SparkR. In the bi n folder, when you download Spark, you will find the various ways of starting these shells.

Simply run spark- shel l (for Scala), spark- sql , pyspark, and sparkR.

After you've finished your application and created a package or script to run, spark- submi t will become your best friend to submit this job to a cluster.

Launching Applications

The most common way for running Spark Applications is through spark- submi t . Previously in this chapter, we showed you how to run spark- submi t ; you simply specify your options, the application JAR or script, and the relevant arguments:

```
. /bi n/spark- submi t \
- - cl ass <mai n- cl ass> \
- - mast er <mast er- url > \
- - depl oy- mode <depl oy- mode> \
- - conf <key>=<val ue> \
  . . . # ot her opt i ons
  <appl i cat i on- j ar- or- scri pt > \
  [ appl i cat i on- argument s]
```

You can always specify whether to run in client or cluster mode when you submit a Spark job with spark- submi t . However, you should almost always favor running in cluster mode (or in client mode on the cluster itself) to reduce latency between the executors and the driver.

When submitting applciations, pass a *.py* file in the place of a *.jar*, and add Python . zi p, . egg, or . py to the search path with - - py- f i l es.

For reference, Table 16-1 lists all of the available spark- submi t options, including those that are particular to some cluster managers. To enumerate all these options yourself, run sparksubmi t with - - hel p.

Table 16-1. Spark submit help text

Parameter	Description
- - mast er MASTER_URL	spark://host:port, mesos://host:port, yarn, or local
- - depl oymode DEPLOY_MODE	Whether to launch the driver program locally (“client”) or on one of the worker machines inside the cluster (“cluster”) (Default: client)
- - cl ass CLASS_NAME	Your application’s main class (for Java / Scala apps).
- - name NAME	A name of your application.
- - j ars J ARS	Comma-separated list of local JARs to include on the driver and executor classpaths.
- - packages	Comma-separated list of Maven coordinates of JARs to include on the driver and executor classpaths. Will search the local Maven repo, then Maven Central and any additional remote repositories given by - - reposi t ori es. The format for the coordinates should be groupI d: art i f act I d: versi on.
- - excl ude-packages	Comma-separated list of groupI d: art i f act I d, to exclude while resolving the dependencies provided in - - packages to avoid dependency conflicts.
- reposi t ori es	Comma-separated list of additional remote repositories to search for the Maven coordinates given with - - packages.
- - py- f i l es PY_FI LES	Comma-separated list of *.zip*, *.egg*, or *.py* files to place on the PYTHONPATH for Python apps.
- - f i l es FI LES	Comma-separated list of files to be placed in the working directory of each executor.
- - conf PROP=VALUE	Arbitrary Spark configuration property.

- propert i esf i l e FI LE	Path to a file from which to load extra properties. If not specified, this will look for conf /spark- def aul t s. conf .
- - dri ver- memory MEM	Memory for driver (e.g., 1000M, 2G) (Default: 1024M).
- - dri verj ava- opt i ons	Extra Java options to pass to the driver.
- - dri ver- l i brary- pat h	Extra library path entries to pass to the driver.
- - dri vercl ass- pat h	Extra class path entries to pass to the driver. Note that JARs added with - - j ars are automatically included in the classpath.
- - execut ormemory MEM	Memory per executor (e.g., 1000M, 2G) (Default: 1G).
- - proxy- user NAME	User to impersonate when submitting the application. This argument does not work with - pri nci pal / - - keyt ab.
- - hel p, - h	Show this help message and exit.
- - verbose, v	Print additional debug output.
- - versi on	Print the version of current Spark.

There are some deployment-specific configurations as well (see Table 16-2).

Table 16-2. Deployment Specific Configurations

Table 16-2. Deployment Specific Configurations

Cluster Managers	Modes	Conf	Description
Standalone	Cluster	- - dri ver- cores NUM	Cores for driver (Default: 1).
Standalone/Mesos	Cluster	- - supervi se	If given, restarts the driver on failure.
Standalone/Mesos	Cluster	- - ki l l SUBMI SSI ON_I D	If given, kills the driver specified.
Standalone/Mesos	Cluster	- - st at us SUBMI SSI ON_I D	If given, requests the status of the driver specified.

Standalone/Mesos	Either	- - t ot al execut orcores NUM	Total cores for all executors.
Standalone/YARN	Either	- - execut orcores NUM1	Number of cores per executor. (Default: 1 in YARN mode or all available cores on the worker in standalone mode)
YARN	Either	- - dri vercores NUM	Number of cores used by the driver, only in cluster mode (Default: 1).
YARN	Either	queue QUEUE_NAME	The YARN queue to submit to (Default: “default”).
YARN	Either	- - numexecut ors NUM	Number of executors to launch (Default: 2). If dynamic allocation is enabled, the initial number of executors will be at least NUM.
YARN	Either	- - archi ves ARCHI VES	Comma-separated list of archives to be extracted into the working directory of each executor.
YARN	Either	- - pri nci pal PRI NCI PAL	Principal to be used to log in to KDC, while running on secure HDFS.
YARN	Either	- - keyt ab KEYTAB	The full path to the file that contains the keytab for the principal specified above. This keytab will be copied to the node running the Application Master via the Secure Distributed Cache, for renewing the login tickets and the delegation tokens periodically.

Application Launch Examples

We already covered some local-mode application examples previously in this chapter, but it’s worth looking at how we use some of the aforementioned options, as well. Spark also includes several examples and demonstration applications in the *examples* directory that is included when you download Spark. If you’re stuck on how to use certain parameters, simply try them first on your local machine and use the SparkPi class as the main class:

```
. /bi n/spark- submi t \
- - cl ass org. apache. spark. exampl es. SparkPi \
- - mast er spark: //207. 184. 161. 138: 7077 \
- - execut or- memory 20G \  - - t ot al - execut or- cores 100 \   repl ace/wi t h/pat
  h/t o/exampl es. j ar \
  1000
```

The following snippet does the same for Python. You run it from the Spark directory and this will allow you to submit a Python application (all in one script) to the standalone cluster manager. You can also set the same executor limits as in the preceding example:

```
. /bi n/spark- submi t \
- - mast er spark: //207. 184. 161. 138: 7077 \  exampl es/src/mai n/pyt hon/pi .
  py \
  1000
```

You can change this to run in local mode as well by setting the master to l ocal or l ocal [*] to run on all the cores on your machine. You will also need to change the /pat h/t o/exampl es. j ar to the relevant Scala and Spark versions you are running.

Configuring Applications

Spark includes a number of different configurations, some of which we covered in Chapter 15. There are many different configurations, depending on what you're hoping to achieve. This section covers those very details. For the most part, this information is included for reference and is probably worth skimming only, unless you're looking for something in particular. The majority of configurations fall into the following categories:

- Application properties
- Runtime environment
- Shuffle behavior
- Spark UI
- Compression and serialization
- Memory management
- Execution behavior
- Networking
- Scheduling
- Dynamic allocation
- Security
- Encryption
- Spark SQL
- Spark streaming
- SparkR

Spark provides three locations to configure the system:

- Spark properties control most application parameters and can be set by using a SparkConf object
- Java system properties
- Hardcoded configuration files

There are several templates that you can use, which you can find in the */conf* directory available in the root of the Spark home folder. You can set these properties as hardcoded variables in your applications or by specifying them at runtime. You can use environment variables to set permachine settings, such as the IP address, through the conf /spark- env. sh script on each node.
Lastly, you can configure logging through l og4j . propert i es.

The SparkConf

The SparkConf manages all of our application configurations. You create one via the i mport statement, as shown in the example that follows. After you create it, the SparkConf is immutable for that specific Spark Application:

```
// i n Scal a i mport org. apache. spark. SparkConf val conf = new SparkConf ( ) . set Mast er( "l ocal
[ 2] ") . set AppName( "Def i ni t i veGui de")   . set ( "some. conf ", "t o. some. val ue")

# i n Pyt hon f rom pyspark i mport SparkConf conf = SparkConf ( ) . set Mast er( "l ocal [ 2]
") . set AppName( "Def i ni t i veGui de") \
  . set ( "some. conf ", "t o. some. val ue")
```

You use the SparkConf to configure individual Spark Applications with Spark properties. These Spark properties control how the Spark Application runs and how the cluster is configured. The example that follows configures the local cluster to have two threads and specifies the application name that shows up in the Spark UI.

You can configure these at runtime, as you saw previously in this chapter through command-line arguments. This is helpful when starting a Spark Shell that will automatically include a basic Spark Application for you; for instance:

```
. /bi n/spark- submi t - - name "Def i ni t i veGui de" - - mast er l ocal [ 4] . . .
```

Of note is that when setting time duration-based properties, you should use the following format:

- 25ms (milliseconds)
- 5s (seconds)
- 10m or 10min (minutes)
- 3h (hours)
- 5d (days)
- 1y (years)

Application Properties

Application properties are those that you set either from spark- submi t or when you create your Spark Application. They define basic application metadata as well as some execution characteristics. Table 16-3 presents a list of current application properties.

Table 16-3. Application properties

Property name	Default	Meaning
spark. app. name	(none)	The name of your application. This will appear in the UI and in log data.
spark. dri ver. cores	1	Number of cores to use for the driver process, only in cluster mode.
spark. dri ver. maxResul t Si ze	1g	Limit of total size of serialized results of all partitions for each Spark action (e.g., collect). Should be at least 1M, or 0 for unlimited. Jobs will be aborted if the total size exceeds this limit. Having a high limit can cause Out Of MemoryErrors in the driver (depends on spark. dri ver. memory and memory overhead of objects in JVM). Setting a proper limit can protect the driver from Out Of MemoryErrors.
spark. dri ver. memory	1g	Amount of memory to use for the driver process, where SparkCont ext is initialized. (e.g. 1g, 2g). Note: in client mode, this must not be set through the SparkConf directly in your application, because the driver JVM has already started at that point. Instead, set this through the - - dri ver- memory command-line option or in your default properties file.
spark. execut or. memory	1g	Amount of memory to use per executor process (e.g., 2g, 8g).
spark. ext raLi st eners	(none)	A comma-separated list of classes that implement SparkLi st ener; when initializing SparkCont ext , instances of these classes will be created and registered with Spark's listener bus. If a class has a single-argument constructor that accepts a SparkConf , that constructor will be called; otherwise, a zero-argument constructor will be called. If no valid constructor can be found, the SparkCont ext creation will fail with an exception.

spark. l ogConf	FALSE	Logs the effective SparkConf as INFO when a SparkCont ext is started.
spark. mast er	(none)	The cluster manager to connect to. See the list of allowed master URLs.
spark. submi t . depl oyMode	(none)	The deploy mode of the Spark driver program, either “client” or “cluster,” which means to launch driver program locally (“client”) or remotely (“cluster”) on one of the nodes inside the cluster.
spark. l og. cal l erCont ext	(none)	Application information that will be written into Yarn RM log/HDFS audit log when running on Yarn/HDFS. Its length depends on the Hadoop configuration hadoop. cal l er. cont ext . max.size. It should be concise, and typically can have up to 50 characters.
spark. dri ver. supervi se	FALSE	If true, restarts the driver automatically if it fails with a non-zero exit status. Only has effect in Spark standalone mode or Mesos cluster deploy mode.

You can ensure that you’ve correctly set these values by checking the application’s web UI on port 4040 of the driver on the “Environment” tab. Only values explicitly specified through *sparkdefaults.conf*, SparkConf , or the command line will appear. For all other configuration properties, you can assume the default value is used.

Runtime Properties

Although less common, there are times when you might also need to configure the runtime environment of your application. Due to space limitations, we cannot include the entire configuration set here. Refer to the relevant table on the Runtime Environment in the Spark documentation. These properties allow you to configure extra classpaths and python paths for both drivers and executors, Python worker configurations, as well as miscellaneous logging properties.

Execution Properties

These configurations are some of the most relevant for you to configure because they give you finer-grained control on actual execution. Due to space limitations, we cannot include the entire configuration set here. Refer to the relevant table on Execution Behavior in the Spark documentation. The most common configurations to change are spark. execut or. cores (to control the number of available cores) and spark. f i l es. maxPart i t i onByt es (maximum partition size when reading files).

Configuring Memory Management

There are times when you might need to manually manage the memory options to try and optimize your applications. Many of these are not particularly relevant for end users because they involve a lot of legacy concepts or fine-grained controls that were obviated in Spark 2.X because of automatic memory management. Due to space limitations, we cannot include the entire configuration set here. Refer to the relevant table on Memory Management in the Spark documentation.

Configuring Shuffle Behavior

We've emphasized how shuffles can be a bottleneck in Spark jobs because of their high communication overhead. Therefore there are a number of low-level configurations for controlling shuffle behavior. Due to space limitations, we cannot include the entire configuration set here. Refer to the relevant table on Shuffle Behavior in the Spark documentation.

Environmental Variables

You can configure certain Spark settings through environment variables, which are read from the *conf/spark-env.sh* script in the directory where Spark is installed (or *conf/spark-env.cmd* on Windows). In Standalone and Mesos modes, this file can give machine-specific information such as hostnames. It is also sourced when running local Spark Applications or submission scripts.

Note that *conf/spark-env.sh* does not exist by default when Spark is installed. However, you can copy *conf/spark-env.sh.template* to create it. Be sure to make the copy executable.

The following variables can be set in *spark-env.sh*:

J AVA_HOME

Location where Java is installed (if it's not on your default PATH).

PYSPARK_PYTHON

Python binary executable to use for PySpark in both driver and workers (default is pyt hon2. 7 if available; otherwise, pyt hon). Property spark. pyspark. pyt hon takes precedence if it is set.

PYSPARK_DRI VER_PYTHON

Python binary executable to use for PySpark in driver only (default is PYSPARK_PYTHON). Property spark. pyspark. dri ver. pyt hon takes precedence if it is set.

SPARKR_DRI VER_R

R binary executable to use for SparkR shell (default is R). Property spark. r. shel l . command takes precedence if it is set.

SPARK_LOCAL_I P

IP address of the machine to which to bind.

SPARK_PUBLI C_DNS

Hostname your Spark program will advertise to other machines.

In addition to the variables ust listed, there are also options for setting up the Spark standalone cluster scripts, such as number of cores to use on each machine and maximum memory. Because *spark-env.sh* is a shell script, you can set some of these programmatically; for example, you might compute SPARK_LOCAL_I P by looking up the IP of a specific network interface.

NOTE

When running Spark on YARN in cluster mode, you need to set environment variables by using the spark. yarn. appMast erEnv. [*Envi ronment Vari abl eName*] property in your *conf/spark-defaults.conf* file. Environment variables that are set in *spark-env.sh* will not be reflected in the YARN Application Master process in cluster mode. See the YARN-related Spark Properties for more information.

Job Scheduling Within an Application

Within a given Spark Application, multiple parallel jobs can run simultaneously if they were submitted from separate threads. By job, in this section, we mean a Spark action and any tasks that need to run to evaluate that action. Spark's scheduler is fully thread-safe and supports this use case to enable applications that serve multiple requests (e.g., queries for multiple users).

By default, Spark's scheduler runs jobs in *FIFO* fashion. If the jobs at the head of the queue don't need to use the entire cluster, later jobs can begin to run right away, but if the jobs at the head of the queue are large, later jobs might be delayed significantly.

It is also possible to configure fair sharing between jobs. Under fair sharing, Spark assigns tasks between jobs in a round-robin fashion so that all jobs get a roughly equal share of cluster resources. This means that short jobs submitted while a long job is running can begin receiving resources right away and still achieve good response times without waiting for the long job to finish. This mode is best for multiuser settings.

To enable the fair scheduler, set the spark. schedul er. mode property to FAI R when configuring a SparkCont ext .

The fair scheduler also supports grouping jobs into pools, and setting different scheduling options, or weights, for each pool. This can be useful to create a high-priority pool for more important jobs or to group the jobs of each user together and give users equal shares regardless of how many concurrent jobs they have instead of giving jobs equal shares. This approach is modeled after the Hadoop Fair Scheduler.

Without any intervention, newly submitted jobs go into a default pool, but jobs pools can be set by adding the spark. schedul er. pool local property to the SparkCont ext in the thread that's submitting them. This is done as follows (assuming sc is your SparkCont ext : sc. set Local Propert y("spark. schedul er. pool ", "pool 1")

After setting this local property, all jobs submitted within this thread will use this pool name. The setting is per-thread to make it easy to have a thread run multiple jobs on behalf of the same user. If you'd like to clear the pool that a thread is associated with, set it to null.

Conclusion

This chapter covered a lot about Spark Applications; we learned how to write, test, run, and configure them in all of Spark's languages. In Chapter 17, we talk about deploying and the cluster management options you have when it comes to running Spark Applications.

Chapter 17. Deploying Spark

This chapter explores the infrastructure you need in place for you and your team to be able to run Spark Applications:

- Cluster deployment choices
- Spark's different cluster managers
- Deployment considerations and configuring deployments

For the most, part Spark should work similarly with all the supported cluster managers; however, customizing the setup means understanding the intricacies of each of the cluster management systems. The hard part is deciding on the cluster manager (or choosing a managed service). Although we would be happy to include all the minute details about how you can configure different cluster with different cluster managers, it's simply impossible for this book to provide hyper-specific details for every situation in every single enviroment. The goal of this chapter, therefore, is not to discuss each of the cluster managers in full detail, but rather to look at their fundamental differences and to provide a reference for a lot of the material already available on the Spark website. Unfortunately, there is no easy answer to "which is the easiest cluster manager to run" because it varies so much by use case, experience, and resources. The Spark documentation site offers a lot of detail about deploying Spark with actionable examples. We do our best to discuss the most relevant points.

As of this writing, Spark has three officially supported cluster managers:

- Standalone mode
- Hadoop YARN
- Apache Mesos

These cluster managers maintain a set of machines onto which you can deploy Spark Applications. Naturally, each of these cluster managers has an opinionated view toward management, and so there are trade-offs and semantics that you will need to keep in mind. However, they all run Spark applications the same way (as covered in Chapter 16). Let's begin with the first point: where to deploy your cluster.

Where to Deploy Your Cluster to Run Spark Applications

There are two high-level options for where to deploy Spark clusters: deploy in an on-premises cluster or in the public cloud. This choice is consequential and is therefore worth discussing.

On-Premises Cluster Deployments

Deploying Spark to an on-premises cluster is sometimes a reasonable option, especially for organizations that already manage their own datacenters. As with everything else, there are tradeoffs to this approach. An on-premises cluster gives you full control over the hardware used, meaning you can optimize performance for your specific workload. However, it also introduces some challenges, especially when it comes to data analytics workloads like Spark. First, with onpremises deployment, your cluster is fixed in size, whereas the resource demands of data analytics workloads are often elastic. If you make your cluster too small, it will be hard to launch the occasional very large analytics query or training job for a new machine learning model, whereas if you make it large, you will have resources sitting idle. Second, for on-premises clusters, you need to select and operate your own storage system, such as a Hadoop file system or scalable key-value store. This includes setting up georeplication and disaster recovery if required.

If you are going to deploy on-premises, the best way to combat the resource utilization problem is to use a cluster manager that allows you to run many Spark applications and dynamically reassign resources between them, or even allows non-Spark applications on the same cluster. All of Spark's supported cluster managers allow multiple concurrent applications, but YARN and Mesos have better support for dynamic sharing and also additionally support non-Spark workloads. Handling resource sharing is likely going to be the biggest difference your users see day to day with Spark on-premise versus in the cloud: in public clouds, it's easy to give each application its own cluster of exactly the required size for just the duration of that job.

For storage, you have several different options, but covering all the trade-offs and operational details in depth would probably require its own book. The most common storage systems used for Spark are distributed file systems such as Hadoop's HDFS and key-value stores such as Apache Cassandra. Streaming message bus systems such as Apache Kafka are also often used for ingesting data. All these systems have varying degrees of support for management, backup, and georeplication, sometimes built into the system and sometimes only through third-party commercial tools. Before choosing a storage option, we recommend evaluating the performance of its Spark connector and evaluating the available management tools.

Spark in the Cloud

While early big data systems were designed for on-premises deployment, the cloud is now an increasingly common platform for deploying Spark. The public cloud has several advantages when it comes to big data workloads. First, resources can be launched and shut down elastically, so you can run that occasional "monster" job that takes hundreds of machines for a few hours without having to pay for them all the time. Even for normal operation, you can choose a different type of machine and cluster size for each application to optimize its cost performance— for example, launch machines with Graphics Processing Units (GPUs) just for your deep learning jobs. Second, public clouds include low-cost, georeplicated storage that makes it easier to manage large amounts of data.

Many companies looking to migrate to the cloud imagine they'll run their applications in the same way that they run their on-premises clusters. All the major cloud providers (Amazon Web Services [AWS], Microsoft Azure, Google Cloud Platform [GCP], and IBM Bluemix) include managed Hadoop clusters for their customers, which provide HDFS for storage as well as Apache Spark. This is actually *not* a great way to run Spark in the cloud, however, because by using a fixed-size cluster and file system, you are not going to be able to take advantage of elasticity. Instead, it is generally a better idea to use global storage systems that are decoupled from a specific cluster, such as Amazon S3, Azure Blob Storage, or Google Cloud Storage and spin up machines dynamically for each Spark workload. With decoupled compute and storage, you will be able to pay for computing resources only when needed, scale them up dynamically, and mix different hardware types. Basically, keep in mind that running Spark in the cloud need not mean migrating an on-premises installation to virtual machines: you can run Spark natively against cloud storage to take full advantage of the cloud's elasticity, cost-saving benefit, and management tools without having to manage an on-premise computing stack within your cloud environment.

Several companies provide "cloud-native" Spark-based services, and all installations of Apache Spark can of course connect to cloud storage. Databricks, the company started by the Spark team from UC Berkeley, is one example of a service provider built specifically for Spark in the cloud.
Databricks provides a simple way to run Spark workloads without the heavy baggage of a Hadoop installation. The company provides a number of features for running Spark more efficiently in the cloud, such as auto-scaling, auto-termination of clusters, and optimized connectors to cloud storage, as well as a collaborative environment for working on notebooks and standalone jobs. The company also provides a free Community Edition for learning Spark where you can run notebooks on a small cluster and share them live with others. A fun fact is that this *entire book* was written using the free Community Edition of Databricks, because we found the integrated Spark notebooks, live collaboration, and cluster management the easiest way to produce and test this content.

If you run Spark in the cloud, much of the content in this chapter might not be relevant because you can often create a separate, short-lived Spark cluster for each job you execute. In that case, the standalone cluster manager is likely the easiest to use. However, you may still want to read this content if you'd like to share a longer-lived cluster among many applications, or to install Spark on virtual machines yourself.

Cluster Managers

Unless you are using a high-level managed service, you will have to decide on the cluster manager to use for Spark. Spark supports three aforementioned cluster managers: standalone clusters, Hadoop YARN, and Mesos. Let's review each of these.

Standalone Mode

Spark's standalone cluster manager is a lightweight platform built specifically for Apache Spark workloads. Using it, you can run multiple Spark Applications on the same cluster. It also provides simple interfaces for doing so but can scale to large Spark workloads. The main disadvantage of the standalone mode is that it's more limited than the other cluster managers—in particular, your cluster can *only* run Spark. It's probably the best starting point if you just want to quickly get Spark running on a cluster, however, and you do not have experience using YARN or Mesos.

Starting a standalone cluster

Starting a standalone cluster requires provisioning the machines for doing so. That means starting them up, ensuring that they can talk to one another over the network, and getting the version of Spark you would like to run on those sets of machines. After that, there are two ways to start the cluster: by hand or using built-in launch scripts.

Let's first launch a cluster by hand. The first step is to start the master process on the machine that we want that to run on, using the following command:

```
$SPARK_HOME/sbi n/st art - mast er. sh
```

When we run this command, the cluster manager master process will start up on that machine. Once started, the master prints out a spark: //HOST: PORT URI. You use this when you start each of the worker nodes of the cluster, and you can use it as the master argument to your SparkSession on application initialization. You can also find this URI on the master's web UI, which is *http://master-ip-address:8080* by default. With that URI, start the worker nodes by logging in to each machine and running the following script using the URI you just received from the master node. The master machine must be available on the network of the worker nodes you are using, and the port must be open on the master node, as well:

```
$SPARK_HOME/sbi n/st art - sl ave. sh <mast er- spark- URI >
```

As soon as you've run that on another machine, you have a Spark cluster running! This process is naturally a bit manual; thankfully there are scripts that can help to automate this process.

Cluster launch scripts

You can configure cluster launch scripts that can automate the launch of standalone clusters. To do this, create a file called *conf/slaves* in your Spark directory that will contain the hostnames of all the machines on which you intend to start Spark workers, one per line. If this file does not exist, everything will launch locally. When you go to actually start the cluster, the master machine will access each of the worker machines via Secure Shell (SSH). By default, SSH is run in parallel and requires that you configure password-less (using a private key) access. If you do not

have a password-less setup, you can set the environment variable SPARK_SSH_FOREGROUND and serially provide a password for each worker.

After you set up this file, you can launch or stop your cluster by using the following shell scripts, based on Hadoop's deploy scripts, and available in $SPARK_HOME/sbi n:

$SPARK_HOME/sbi n/st art - mast er. sh

Starts a master instance on the machine on which the script is executed.

$SPARK_HOME/sbi n/st art - sl aves. sh

Starts a slave instance on each machine specified in the *conf/slaves* file.

$SPARK_HOME/sbi n/st art - sl ave. sh

Starts a slave instance on the machine on which the script is executed.

$SPARK_HOME/sbi n/st art - al l . sh

Starts both a master and a number of slaves as described earlier.

$SPARK_HOME/sbi n/st op- mast er. sh

Stops the master that was started via the *bin/start-master.sh* script.

$SPARK_HOME/sbi n/st op- sl aves. sh

Stops all slave instances on the machines specified in the *conf/slaves* file.

$SPARK_HOME/sbi n/st op- al l . sh

Stops both the master and the slaves as described earlier.

Standalone cluster configurations

Standalone clusters have a number of configurations that you can use to tune your application. These control everything from what happens to old files on each worker for terminated applications to the worker's core and memory resources. These are controlled via environment variables or via application properties. Due to space limitations, we cannot include the entire configuration set here. Refer to the relevant table on Standalone Environment Variables in the Spark documentation.

Submitting applications

After you create the cluster, you can submit applications to it using the spark: // URI of the master. You can do this either on the master node itself or another machine using spark- submi t . There are some specific command-line arguments for standalone mode, which we covered in "Launching Applications".

Spark on YARN

Hadoop YARN is a framework for job scheduling and cluster resource management. Even though Spark is often (mis)classified as a part of the "Hadoop Ecosystem," in reality, Spark has little to do with Hadoop. Spark does natively support the Hadoop YARN cluster manager but it requires nothing from Hadoop itself.

You can run your Spark jobs on Hadoop YARN by specifying the master as YARN in the spark-submi t command-line arguments. Just like with standalone mode, there are a number of knobs that you are able to tune according to what you would like the cluster to do. The number of knobs is naturally larger than that of Spark's standalone mode because Hadoop YARN is a generic scheduler for a large number of different execution frameworks.

Setting up a YARN cluster is beyond the scope of this book, but there are some great books on the topic as well as managed services that can simplify this experience.

Submitting applications

When submitting applications to YARN, the core difference from other deployments is that -mast er will become yarn as opposed the master node IP, as it is in standalone mode. Instead, Spark will find the YARN configuration files using the environment variable HADOOP_CONF_DI R or YARN_CONF_DI R. Once you have set those environment variables to your Hadoop installation's configuration directory, you can just run spark- submi t like we saw in Chapter 16.

> **NOTE**
>
> There are two deployment modes that you can use to launch Spark on YARN. As discussed in previous chapters, cluster mode has the spark driver as a process managed by the YARN cluster, and the client can exit after creating the application. In client mode, the driver will run in the client process and therefore YARN will be responsible only for granting executor resources to the application, not maintaining the master node. Also of note is that in cluster mode, Spark doesn't necessarily run on the same machine on which you're executing. Therefore libraries and external jars must be distributed manually or through the - - j ars command-line argument.

There are a few YARN-specific properties that you can set by using spark- submi t . These allow you to control priority queues and things like keytabs for security. We covered these in "Launching Applications" in Chapter 16.

Configuring Spark on YARN Applications

Deploying Spark as YARN applications requires you to understand the variety of different configurations and their implications for your Spark applications. This section covers some best practices for basic configurations and includes references to some of the important configuration for running your Spark applications.

Hadoop configurations

If you plan to read and write from HDFS using Spark, you need to include two Hadoop configuration files on Spark's classpath: *hdfs-site.xml*, which provides default behaviors for the HDFS client; and *core-site.xml*, which sets the default file system name. The location of these configuration files varies across Hadoop versions, but a common location is inside of */etc/hadoop/conf*. Some tools create these configurations on the fly, as well, so it's important to understand how your managed service might be deploying these, as well.

To make these files visible to Spark, set HADOOP_CONF_DI R in *$SPARK_HOME/spark-env.sh* to a location containing the configuration files or as an environment variable when you go to sparksubmi t your application.

Application properties for YARN

There are a number of Hadoop-related configurations and things that come up that largely don't have much to do with Spark, just running or securing YARN in a way that influences how Spark runs. Due to space limitations, we cannot include the configuration set here. Refer to the relevant table on YARN Configurations in the Spark documentation.

Spark on Mesos

Apache Mesos is another clustering system that Spark can run on. A fun fact about Mesos is that the project was also started by many of the original authors of Spark, including one of the authors of this book. In the Mesos project's own words:

> *Apache Mesos abstracts CPU, memory, storage, and other compute resources away from machines (physical or virtual), enabling fault-tolerant and elastic distributed systems to easily be built and run effectively.*

For the most part, Mesos intends to be a datacenter scale-cluster manager that manages not just short-lived applications like Spark, but long-running applications like web applications or other resource interfaces. Mesos is the heaviest-weight cluster manager, simply because you might choose this cluster manager only if your organization already has a large-scale deployment of Mesos, but it makes for a good cluster manager nonetheless.

Mesos is a large piece of infrastructure, and unfortunately there's simply too much information for us to cover how to deploy and maintain Mesos clusters. There are many great books on the subject for that, including Dipa Dubhashi and Akhil Das's *Mastering Mesos* (O'Reilly, 2016). The goal here is to bring up some of the considerations that you'll need to think about when running Spark Applications on Mesos.

For instance, one common thing you will hear about Spark on Mesos is fine-grained versus coarse-grained mode. Historically Mesos supported a variety of different modes (fine-grained and coarse-grained), but at this point, it supports only coarse-grained scheduling (fine-grained

has been deprecated). Coarse-grained mode means that each Spark executor runs as a single Mesos task. Spark executors are sized according to the following application properties:

- spark. execut or. memory spark. execut or.
- cores spark. cores. max/spark. execut or.
- cores

Submitting applications

Submitting applications to a Mesos cluster is similar to doing so for Spark's other cluster managers. For the most part you should favor cluster mode when using Mesos. Client mode requires some extra configuration on your part, especially with regard to distributing resources around the cluster.

For instance, in client mode, the driver needs extra configuration information in *spark-env.sh* to work with Mesos.

In *spark-env.sh* set some environment variables: export

```
MESOS_NATI VE_J AVA_LI BRARY=<pat h t o l i bmesos. so>
```

This path is typically *<prefix>/lib/libmesos.so* where the prefix is */usr/local* by default. On Mac OS X, the library is called *libmesos.dylib* instead of *libmesos.so*: export SPARK_EXECUTOR_URI =<URL of spark- 2. 2. 0. t ar. gz upl oaded above>

Finally, set the Spark Application property spark. execut or. uri to <URL of spark2. 2. 0. t ar. gz>. Now, when starting a Spark application against the cluster, pass a mesos: // URL as the master when creating a SparkCont ex, and set that property as a parameter in your SparkConf variable or the initialization of a SparkSessi on:

```
// i n Scal a i mport org. apache. spark. sql .
SparkSessi on val spark = SparkSessi on. bui l der
  . mast er( "mesos: //HOST: 5050")
  . appName( "my app")
  . conf i g( "spark. execut or. uri ", "<pat h t o spark- 2. 2. 0. t ar. gz upl oaded above>")  . get OrCreat
e( )
```

Submitting cluster mode applications is fairly straightforward and follows the same sparksubmi t structure you read about before. We covered these in "Launching Applications".

Configuring Mesos

Just like any other cluster manager, there are a number of ways that we can configure our Spark Applications when they're running on Mesos. Due to space limitations, we cannot include the

entire configuration set here. Refer to the relevant table on Mesos Configurations in the Spark documentation.

Secure Deployment Configurations

Spark also provides some low-level ability to make your applications run more securely, especially in untrusted environments. Note that the majority of this setup will happen outside of Spark. These configurations are primarily network-based to help Spark run in a more secure manner. This means authentication, network encryption, and setting TLS and SSL configurations. Due to space limitations, we cannot include the entire configuration set here.

Refer to the relevant table on Security Configurations in the Spark documentation.

Cluster Networking Configurations

Just as shuffles are important, there can be some things worth tuning on the network. This can also be helpful when performing custom deployment configurations for your Spark clusters when you need to use proxies in between certain nodes. If you're looking to increase Spark's performance, these should not be the first configurations you go to tune, but may come up in custom deployment scenarios. Due to space limitations, we cannot include the entire configuration set here. Refer to the relevant table on Networking Configurations in the Spark documentation.

Application Scheduling

Spark has several facilities for scheduling resources between computations. First, recall that, as described earlier in the book, each Spark Application runs an independent set of executor processes. Cluster managers provide the facilities for scheduling across Spark applications. Second, within each Spark application, multiple jobs (i.e., Spark actions) may be running concurrently if they were submitted by different threads. This is common if your application is serving requests over the network. Spark includes a *fair scheduler* to schedule resources within each application. We introduced this topic in the previous chapter.

If multiple users need to share your cluster and run different Spark Applications, there are different options to manage allocation, depending on the cluster manager. The simplest option, available on all cluster managers, is static partitioning of resources. With this approach, each application is given a maximum amount of resources that it can use, and holds onto those resources for the entire duration. In spark- submi t there are a number of properties that you can set to control the resource allocation of a particular application. Refer to Chapter 16 for more information. In addition, *dynamic allocation* (described next) can be turned on to let applications scale up and down dynamically based on their current number of pending tasks. If, instead, you want users to be able to share memory and executor resources in a fine-grained

manner, you can launch a single Spark Application and use thread scheduling within it to serve multiple requests in parallel.

Dynamic allocation

If you would like to run multiple Spark Applications on the same cluster, Spark provides a mechanism to dynamically adjust the resources your application occupies based on the workload. This means that your application can give resources back to the cluster if they are no longer used, and request them again later when there is demand. This feature is particularly useful if multiple applications share resources in your Spark cluster.

This feature is disabled by default and available on all coarse-grained cluster managers; that is, standalone mode, YARN mode, and Mesos coarse-grained mode. There are two requirements for using this feature. First, your application must set spark. dynami cAl l ocat i on. enabl ed to

t rue. Second, you must set up an external shuffle service on each worker node in the same cluster and set spark. shuf f l e. servi ce. enabl ed to t rue in your application. The purpose of the external shuffle service is to allow executors to be removed without deleting shuffle files written by them. This is set up differently for each cluster manager and is described in the job scheduling configuration. Due to space limitations, we cannot include the configuration set for dynamic allocation. Refer to the relevant table on Dynamic Allocation Configurations.

Miscellaneous Considerations

There several other topics to consider when deploying Spark applications that may affect your choice of cluster manager and its setup. These are just things that you should think about when comparing different deployment options.

One of the more important considerations is the number and type of applications you intend to be running. For instance, YARN is great for HDFS-based applications but is not commonly used for much else. Additionally, it's not well designed to support the cloud, because it expects information to be available on HDFS. Also, compute and storage is largely coupled together, meaning that scaling your cluster involves scaling both storage and compute instead of just one or the other. Mesos does improve on this a bit conceptually, and it supports a wide range of application types, but it still requires pre-provisioning machines and, in some sense, requires buy-in at a much larger scale. For instance, it doesn't really make sense to have a Mesos cluster for only running Spark Applications. Spark standalone mode is the lightest-weight cluster manager and is relatively simple to understand and take advantage of, but then you're going to be building more application management infrastructure that you could get much more easily by using YARN or Mesos.

Another challenge is managing different Spark versions. Your hands are largely tied if you want to try to run a variety of different applications running different Spark versions, and unless you use a well-managed service, you're going to need to spend a fair amount of time either

managing different setup scripts for different Spark services or removing the ability for your users to use a variety of different Spark applications.

Regardless of the cluster manager that you choose, you're going to want to consider how you're going to set up logging, store logs for future reference, and allow end users to debug their applications. These are more "out of the box" for YARN or Mesos and might need some tweaking if you're using standalone.

One thing you might want to consider—or that might influence your decision making—is maintaining a metastore in order to maintain metadata about your stored datasets, such as a table catalog. We saw how this comes up in Spark SQL when we are creating and maintaining tables. Maintaining an Apache Hive metastore, a topic beyond the scope of this book, might be something that's worth doing to facilitate more productive, cross-application referencing to the same datasets.

Depending on your workload, it might be worth considering using Spark's external shuffle service. Typically Spark stores shuffle blocks (shuffle output) on a local disk on that particular node. An external shuffle service allows for storing those shuffle blocks so that they are available to all executors, meaning that you can arbitrarily kill executors and still have their shuffle outputs available to other applications.

Finally, you're going to need to configure at least some basic monitoring solution and help users debug their Spark jobs running on their clusters. This is going to vary across cluster management options and we touch on some of the things that you might want to set up in Chapter 18.

Conclusion

This chapter looked at the world of configuration options that you have when choosing how to deploy Spark. Although most of the information is irrelevant to the majority of users, it is worth mentioning if you're performing more advanced use cases. It might seem fallacious, but there are other configurations that we have omitted that control even lower-level behavior. You can find these in the Spark documentation or in the Spark source code. Chapter 18 talks about some of the options that we have when monitoring Spark Applications.

Chapter 18. Monitoring and Debugging

This chapter covers the key details you need to monitor and debug your Spark Applications. To do this, we will walk through the Spark UI with an example query designed to help you understand how to trace your own jobs through the execution life cycle. The example we'll look at will also help you understand how to debug your jobs and where errors are likely to occur.

The Monitoring Landscape

At some point, you'll need to monitor your Spark jobs to understand where issues are occuring in them. It's worth reviewing the different things that we can actually monitor and outlining some of the options for doing so. Let's review the components we can monitor (see Figure 18-1).

Spark Applications and Jobs

: The first thing you'll want to begin monitoring when either debugging or just understanding better how your application executes against the cluster is the Spark UI and the Spark logs. These report information about the applications currently running at the level of concepts in Spark, such as RDDs and query plans. We talk in detail about how to use these Spark monitoring tools throughout this chapter.

JVM

: Spark runs the executors in individual Java Virtual Machines (JVMs). Therefore, the next level of detail would be to monitor the individual virtual machines (VMs) to better understand how your code is running. JVM utilities such as *jstack* for providing stack traces, *jmap* for creating heap-dumps, *jstat* for reporting time–series statistics, and *jconsole* for visually exploring various JVM properties are useful for those comfortable with JVM internals. You can also use a tool like *jvisualvm* to help profile Spark jobs. Some of this information is provided in the Spark UI, but for very low-level debugging, the aforementioned tools can come in handy.

OS/Machine

: The JVMs run on a host operating system (OS) and it's important to monitor the state of those machines to ensure that they are healthy. This includes monitoring things like CPU, network, and I/O. These are often reported in cluster-level monitoring solutions; however, there are more specific tools that you can use, including *dstat*, *iostat*, and *iotop*.

Cluster

Naturally, you can monitor the cluster on which your Spark Application(s) will run. This might be a YARN, Mesos, or standalone cluster. Usually it's important to have some sort of monitoring solution here because, somewhat obviously, if your cluster is not working, you should probably know pretty quickly. Some popular cluster-level monitoring tools include *Ganglia* and *Prometheus*.

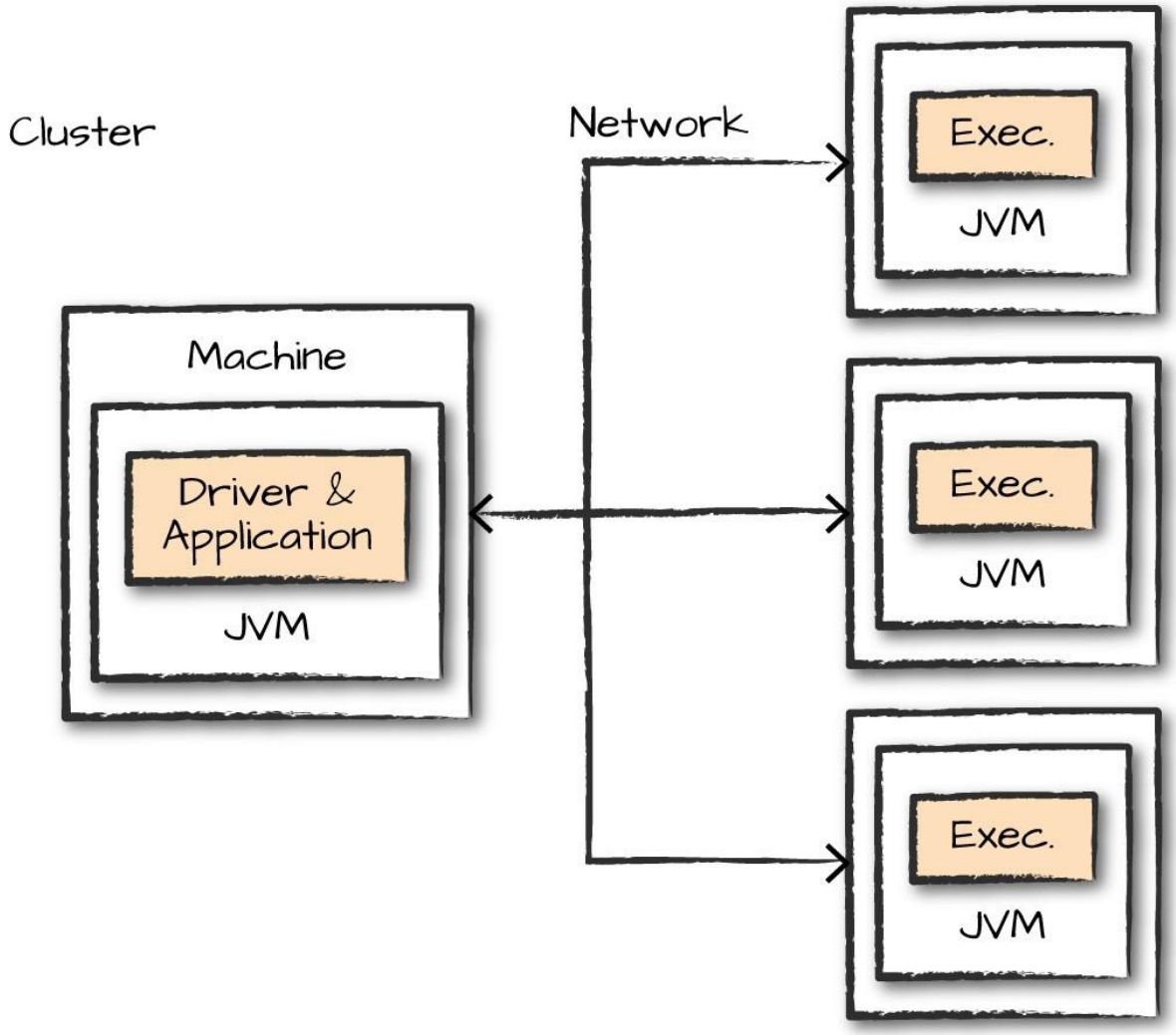


Figure 18-1. Components of a Spark application that you can monitor

What to Monitor

After that brief tour of the monitoring landscape, let's discuss how we can go about monitoring and debugging our Spark Applications. There are two main things you will want to monitor: the *processes* running your application (at the level of CPU usage, memory usage, etc.), and the *query execution* inside it (e.g., jobs and tasks).

Driver and Executor Processes

When you're monitoring a Spark application, you're definitely going to want to keep an eye on the driver. This is where all of the state of your application lives, and you'll need to be sure it's running in a stable manner. If you could monitor only one machine or a single JVM, it would definitely be the driver. With that being said, understanding the state of the executors is also extremely important for monitoring individual Spark jobs. To help with this challenge, Spark has a configurable metrics system based on the Dropwizard Metrics Library. The metrics system is configured via a configuration file that Spark expects to be present at *$SPARK_HOME/conf/metrics.properties*. A custom file location can be specified by changing the spark. met ri cs. conf configuration property. These metrics can be output to a variety of different sinks, including cluster monitoring solutions like Ganglia.

Queries, Jobs, Stages, and Tasks

Although the driver and executor processes are important to monitor, sometimes you need to debug what's going on at the level of a specific query. Spark provides the ability to dive into *queries*, *jobs*, *stages*, and *tasks*. (We learned about these in Chapter 15.) This information allows you to know exactly what's running on the cluster at a given time. When looking for performance tuning or debugging, this is where you are most likely to start.

Now that we know what we want to monitor, let's look at the two most common ways of doing so: the Spark logs and the Spark UI.

Spark Logs

One of the most detailed ways to monitor Spark is through its log files. Naturally, strange events in Spark's logs, or in the logging that you added to your Spark Application, can help you take note of exactly where jobs are failing or what is causing that failure. If you use the application template provided with the book, the logging framework we set up in the template will allow your application logs to show up along Spark's own logs, making them very easy to correlate. One challenge, however, is that Python won't be able to integrate directly with Spark's Javabased logging library. Using Python's l oggi ng module or even simple print statements will still print the results to standard error, however, and make them easy to find.

To change Spark's log level, simply run the following command: spark.

```
sparkCont ext . set LogLevel ( "I NFO")
```

This will allow you to read the logs, and if you use our application template, you can log your own relevant information along with these logs, allowing you to inspect both your own application and Spark. The logs themselves will be printed to standard error when running a local mode application, or saved to files by your cluster manager when running Spark on a

cluster. Refer to each cluster manager's documentation about how to find them—typically, they are available through the cluster manager's web UI.

You won't always find the answer you need simply by searching logs, but it can help you pinpoint the given problem that you're encountering and possibly add new log statements in your application to better understand it. It's also convenient to collect logs over time in order to reference them in the future. For instance, if your application crashes, you'll want to debug why, without access to the now-crashed application. You may also want to ship logs off the machine they were written on to hold onto them if a machine crashes or gets shut down (e.g., if running in the cloud).

The Spark UI

The Spark UI provides a visual way to monitor applications while they are running as well as metrics about your Spark workload, at the Spark and JVM level. Every SparkCont ext running launches a web UI, by default on port 4040, that displays useful information about the application. When you run Spark in local mode, for example, just navigate to *http://localhost:4040* to see the UI when running a Spark Application on your local machine. If you're running multiple applications, they will launch web UIs on increasing port numbers (4041, 4042, …). Cluster managers will also link to each application's web UI from their own UI.

Figure 18-2 shows all of the tabs available in the Spark UI.

Figure 18-2. Spark UI tabs

These tabs are accessible for each of the things that we'd like to monitor. For the most part, each of these should be self-explanatory:

- The Jobs tab refers to Spark jobs.
- The Stages tab pertains to individual stages (and their relevant tasks).
- The Storage tab includes information and the data that is currently cached in our Spark Application.
- The Environment tab contains relevant information about the configurations and current settings of the Spark application.
- The SQL tab refers to our Structured API queries (including SQL and DataFrames).
- The Executors tab provides detailed information about each executor running our application.

Let's walk through an example of how you can drill down into a given query. Open a new Spark shell, run the following code, and we will trace its execution through the Spark UI:

```
# i n Pyt hon spark.
read\
 . opt i on( "header", "t rue") \
 . csv( "/dat a/ret ai l - dat a/al l /onl i ne- ret ai l - dat aset . csv") \
 . repart i t i on( 2) \
 . sel ect Expr( "i nst r( Descri pt i on, ' GLASS' ) >= 1 as i s_gl ass") \
 . groupBy( "i s_gl ass") \
 . count ( ) \
 . col l ect ( )
```

This results in three rows of various values. The code kicks off a SQL query, so let's navigate to the SQL tab, where you should see something similar to Figure 18-3.

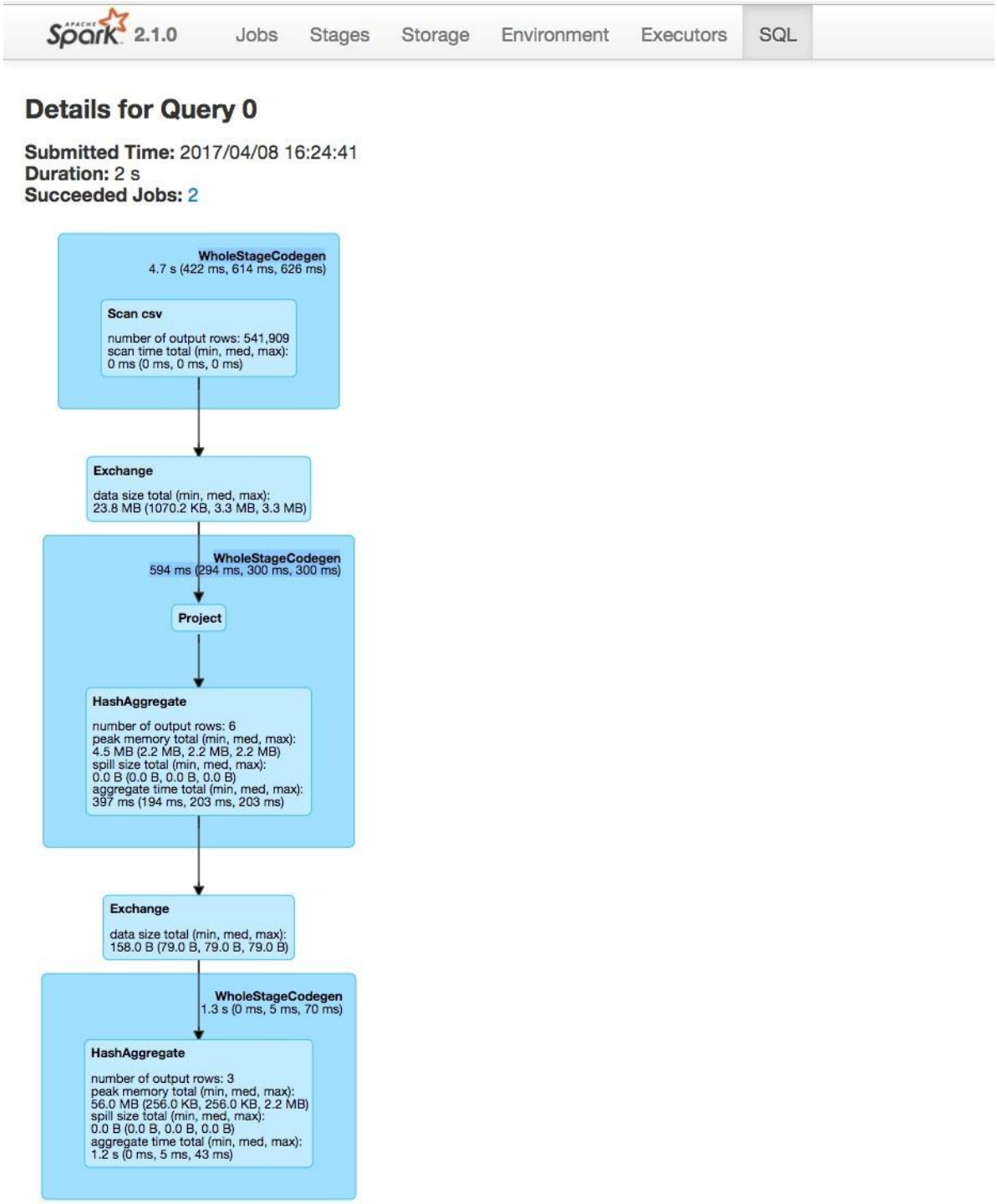


Figure 18-3. The SQL tab

The first thing you see is aggregate statistics about this query:

Submi t t ed Ti me: 2017/04/08 16: 24: 41
Durat i on: 2 s
Succeeded J obs: 2

These will become important in a minute, but first let's take a look at the Directed Acyclic Graph (DAG) of Spark stages. Each blue box in these tabs represent a stage of Spark tasks. The entire group of these stages represent our Spark job. Let's take a look at each stage in detail so that we can better understand what is going on at each level, starting with Figure 18-4.

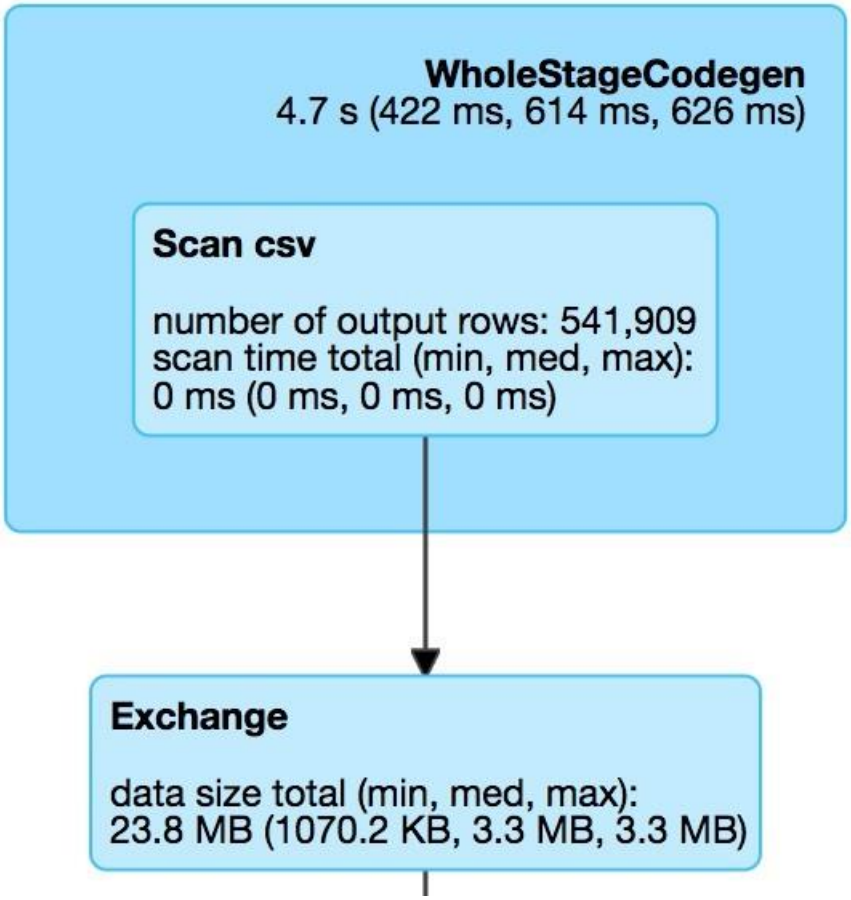


Figure 18-4. Stage one

The box on top, labeled WholeStateCodegen, represents a full scan of the CSV file. The box below that represents a shuffle that we forced when we called repartition. This turned our original dataset (of a yet to be specified number of partitions) into two partitions.

The next step is our projection (selecting/adding/filtering columns) and the aggregation. Notice that in Figure 18-5 the number of output rows is six. This convienently lines up with the number of output rows multiplied by the number of partitions at aggregation time. This is because Spark performs an aggregation for each partition (in this case a hash-based aggregation) before shuffling the data around in preparation for the final stage.

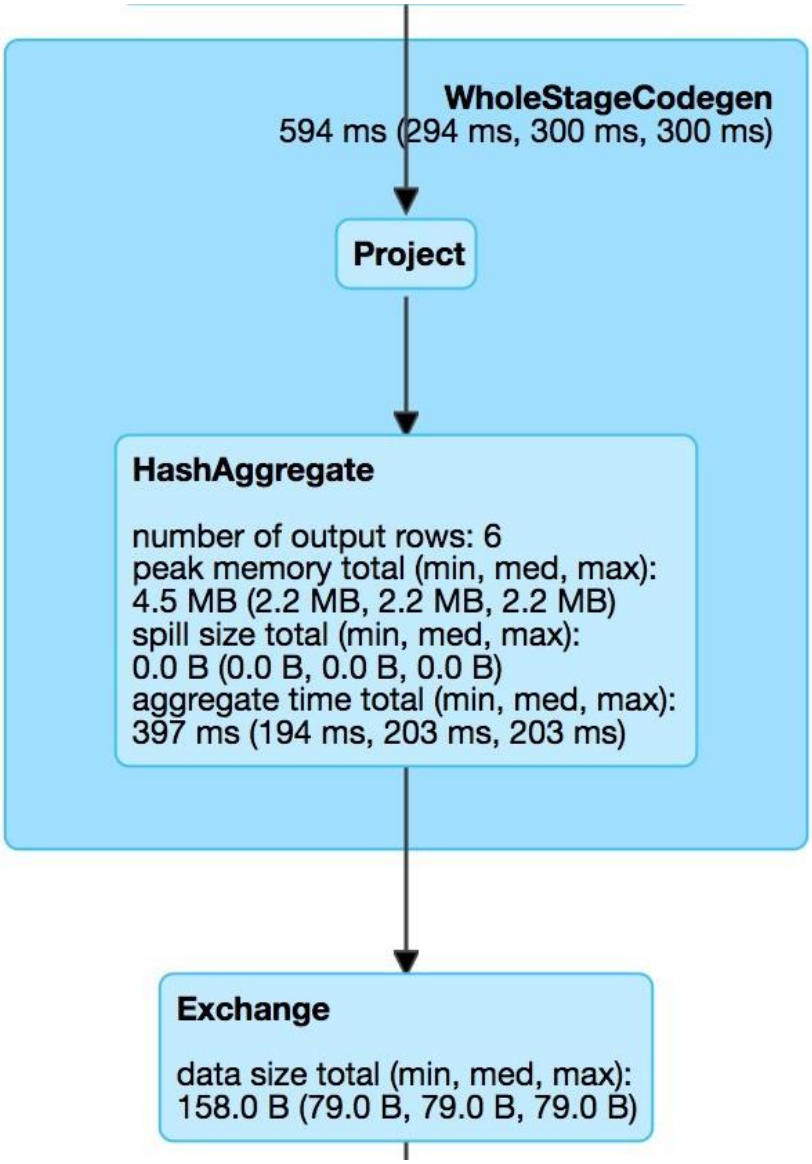


Figure 18-5. Stage two

The last stage is the aggregation of the subaggregations that we saw happen on a per-partition basis in the previous stage. We combine those two partitions in the final three rows that are the output of our total query (Figure 18-6).

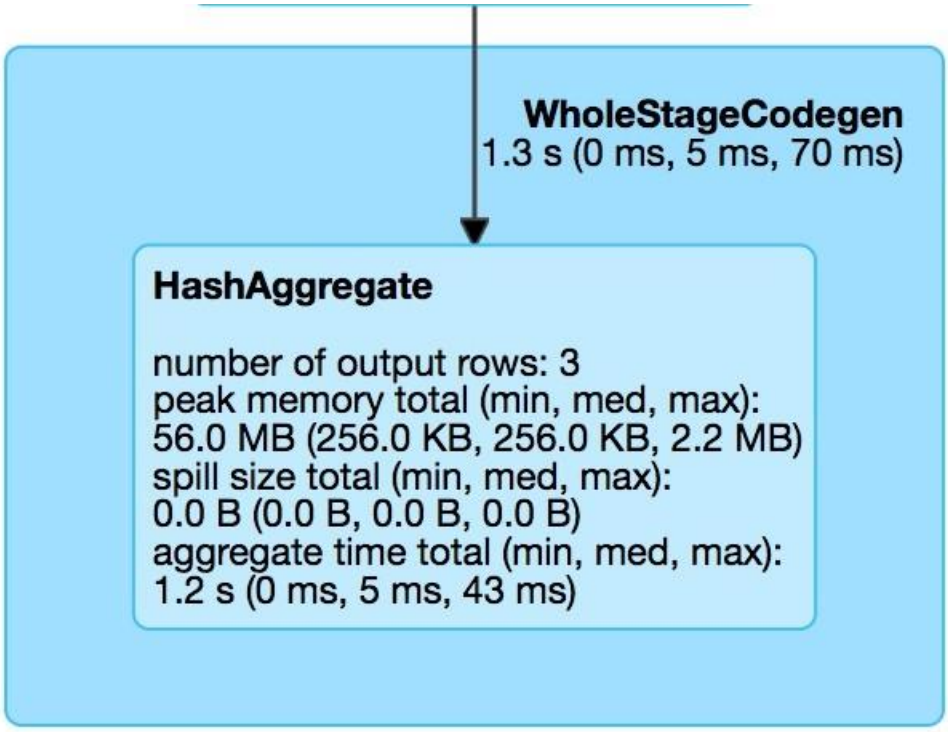


Figure 18-6. Stage three

Let's look further into the job's execution. On the Jobs tab, next to Succeeded Jobs, click 2. As Figure 18-7 demonstrates, our job breaks down into three stages (which corresponds to what we saw on the SQL tab).

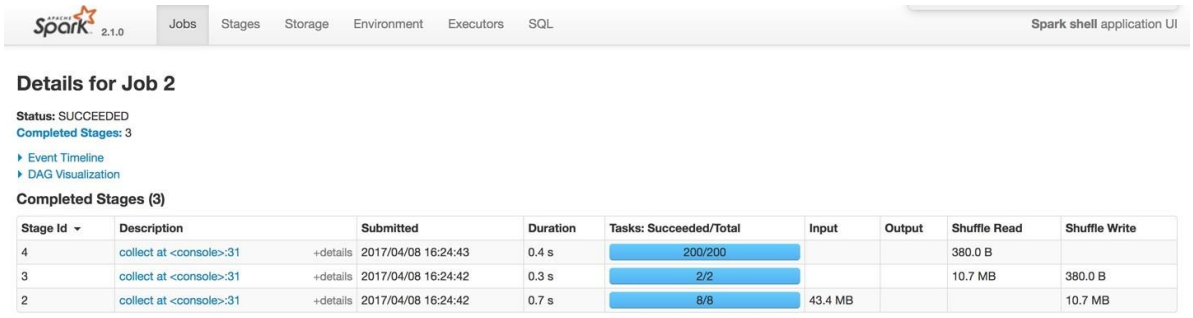


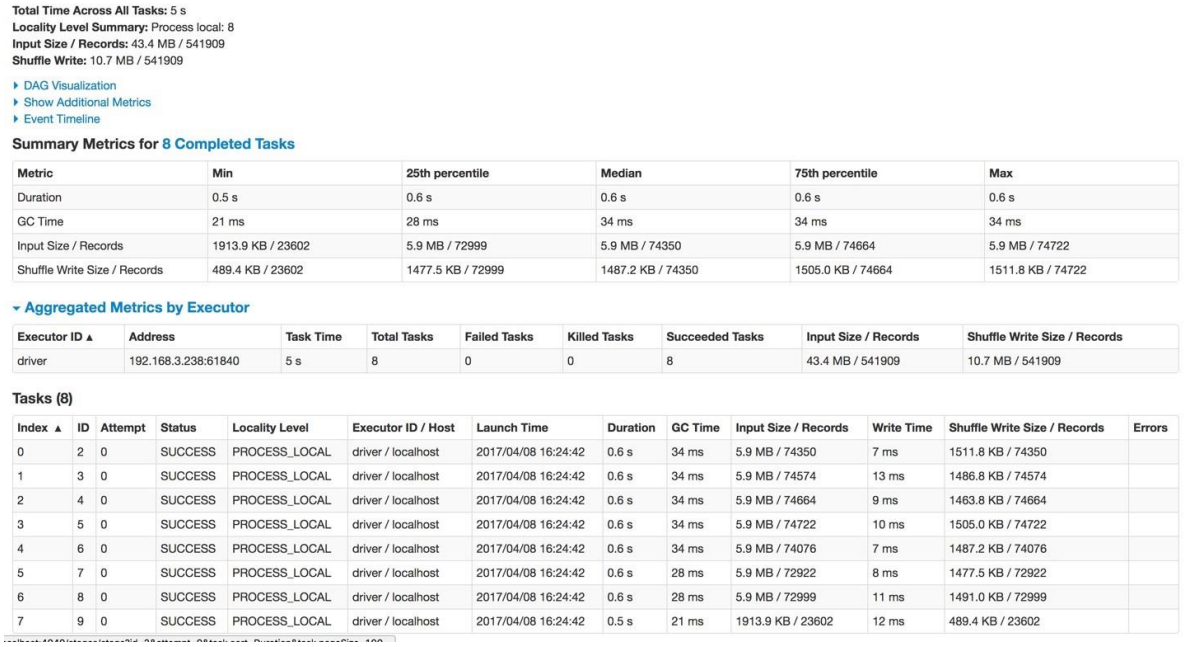
Figure 18-7. The Jobs tab

These stages have more or less the same information as what's shown in Figure 18-6, but clicking the label for one of them will show the details for a given stage. In this example, three stages ran, with eight, two, and then two hundred tasks each. Before diving into the stage detail, let's review why this is the case.

The first stage has eight tasks. CSV files are splittable, and Spark broke up the work to be distributed relatively evenly between the different cores on the machine. This happens at the cluster level and points to an important optimization: how you store your files. The following

stage has two tasks because we explicitly called a repartition to move the data into two partitions. The last stage has 200 tasks because the default shuffle partitions value is 200.

Now that we reviewed how we got here, click the stage with eight tasks to see the next level of detail, as shown in Figure 18-8.



Total Time Across All Tasks: 5 s
Locality Level Summary: Process local: 8
Input Size / Records: 43.4 MB / 541909
Shuffle Write: 10.7 MB / 541909

▸ DAG Visualization
▸ Show Additional Metrics
▸ Event Timeline

Summary Metrics for 8 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	0.5 s	0.6 s	0.6 s	0.6 s	0.6 s
GC Time	21 ms	28 ms	34 ms	34 ms	34 ms
Input Size / Records	1913.9 KB / 23602	5.9 MB / 72999	5.9 MB / 74350	5.9 MB / 74664	5.9 MB / 74722
Shuffle Write Size / Records	489.4 KB / 23602	1477.5 KB / 72999	1487.2 KB / 74350	1505.0 KB / 74664	1511.8 KB / 74722

▾ Aggregated Metrics by Executor

Executor ID ▲	Address	Task Time	Total Tasks	Failed Tasks	Killed Tasks	Succeeded Tasks	Input Size / Records	Shuffle Write Size / Records
driver	192.168.3.238:61840	5 s	8	0	0	8	43.4 MB / 541909	10.7 MB / 541909

Tasks (8)

Index ▲	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Input Size / Records	Write Time	Shuffle Write Size / Records	Errors
0	2	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2017/04/08 16:24:42	0.6 s	34 ms	5.9 MB / 74350	7 ms	1511.8 KB / 74350	
1	3	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2017/04/08 16:24:42	0.6 s	34 ms	5.9 MB / 74574	13 ms	1486.8 KB / 74574	
2	4	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2017/04/08 16:24:42	0.6 s	34 ms	5.9 MB / 74664	9 ms	1463.8 KB / 74664	
3	5	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2017/04/08 16:24:42	0.6 s	34 ms	5.9 MB / 74722	10 ms	1505.0 KB / 74722	
4	6	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2017/04/08 16:24:42	0.6 s	34 ms	5.9 MB / 74076	7 ms	1487.2 KB / 74076	
5	7	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2017/04/08 16:24:42	0.6 s	28 ms	5.9 MB / 72922	8 ms	1477.5 KB / 72922	
6	8	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2017/04/08 16:24:42	0.6 s	28 ms	5.9 MB / 72999	11 ms	1491.0 KB / 72999	
7	9	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2017/04/08 16:24:42	0.5 s	21 ms	1913.9 KB / 23602	12 ms	489.4 KB / 23602	

Figure 18-8. Spark tasks

Spark provides a lot of detail about what this job did when it ran. Toward the top, notice the Summary Metrics section. This provides a synopsis of statistics regarding various metrics. What you want to be on the lookout for is uneven distributions of the values (we touch on this in Chapter 19). In this case, everything looks very consistent; there are no wide swings in the distribution of values. In the table at the bottom, we can also examine on a per-executor basis (one for every core on this particular machine, in this case). This can help identify whether a particular executor is struggling with its workload.

Spark also makes available a set of more detailed metrics, as shown in Figure 18-8, which are probably not relevant to the large majority of users. To view those, click Show Additional Metrics, and then either choose (De)select All or select individual metrics, depending on what you want to see.

You can repeat this basic analysis for each stage that you want to analyze. We leave that as an exercise for the reader.

Other Spark UI tabs

The remaining Spark tabs, Storage, Environment, and Executors, are fairly self-explanatory. The Storage tab shows information about the cached RDDs/DataFrames on the cluster. This can help you see if certain data has been evicted from the cache over time. The Environment tab shows

you information about the Runtime Environment, including information about Scala and Java as well as the various Spark Properties that you configured on your cluster.

Configuring the Spark user interface

There are a number of configurations that you can set regarding the Spark UI. Many of them are networking configurations such as enabling access control. Others let you configure how the Spark UI will behave (e.g., how many jobs, stages, and tasks are stored). Due to space limitations, we cannot include the entire configuration set here. Consult the relevant table on Spark UI Configurations in the Spark documentation.

Spark REST API

In addition to the Spark UI, you can also access Spark's status and metrics via a REST API. This is is available at *http://localhost:4040/api/v1* and is a way of building visualizations and monitoring tools on top of Spark itself. For the most part this API exposes the same information presented in the web UI, except that it doesn't include any of the SQL-related information. This can be a useful tool if you would like to build your own reporting solution based on the information available in the Spark UI. Due to space limitations, we cannot include the list of API endpoints here. Consult the relevant table on REST API Endpoints in the Spark documentation.

Spark UI History Server

Normally, the Spark UI is only available while a SparkContext is running, so how can you get to it after your application crashes or ends? To do this, Spark includes a tool called the Spark History Server that allows you to reconstruct the Spark UI and REST API, provided that the application was configured to save an *event log*. You can find up-to-date information about how to use this tool in the Spark documentation.

To use the history server, you first need to configure your application to store event logs to a certain location. You can do this by by enabling spark. event Log. enabl ed and the event log location with the configuration spark. event Log. di r. Then, once you have stored the events, you can run the history server as a standalone application, and it will automatically reconstruct the web UI based on these logs. Some cluster managers and cloud services also configure logging automatically and run a history server by default.

There are a number of other configurations for the history server. Due to space limitations, we cannot include the entire configuration set here. Refer to the relevant table on Spark History Server Configurations in the Spark documentation.

Debugging and Spark First Aid

The previous sections defined some core "vital signs"—that is, things that we can monitor to check the health of a Spark Application. For the remainder of the chapter we're going to take a

"first aid" approach to Spark debugging: We'll review some signs and symptoms of problems in your Spark jobs, including signs that you might observe (e.g., slow tasks) as well as symptoms from Spark itself (e.g., Out Of MemoryError). There are many issues that may affect Spark jobs, so it's impossible to cover everything. But we will discuss some of the more common Spark issues you may encounter. In addition to the signs and symptoms, we'll also look at some potential treatments for these issues.

Most of the recommendations about fixing issues refer to the configuration tools discussed in Chapter 16.

Spark Jobs Not Starting

This issue can arise frequently, especially when you're just getting started with a fresh deployment or environment.

Signs and symptoms

- Spark jobs don't start.
- The Spark UI doesn't show any nodes on the cluster except the driver.
- The Spark UI seems to be reporting incorrect information.

Potential treatments

This mostly occurs when your cluster or your application's resource demands are not configured properly. Spark, in a distributed setting, does make some assumptions about networks, file systems, and other resources. During the process of setting up the cluster, you likely configured something incorrectly, and now the node that runs the driver cannot talk to the executors. This might be because you didn't specify what IP and port is open or didn't open the correct one. This is most likely a cluster level, machine, or configuration issue. Another option is that your application requested more resources per executor than your cluster manager currently has free, in which case the driver will be waiting forever for executors to be launched.

- Ensure that machines can communicate with one another on the ports that you expect. Ideally, you should open up all ports between the worker nodes unless you have more stringent security constraints.
- Ensure that your Spark resource configurations are correct and that your cluster manager is properly set up for Spark. Try running a simple application first to see if that works. One common issue may be that you requested more memory per executor than the cluster manager has free to allocate, so check how much it is reporting free (in its UI) and your spark- submi t memory configuration.

Errors Before Execution

This can happen when you're developing a new application and have previously run code on this cluster, but now some new code won't work.

Signs and symptoms

- Commands don't run at all and output large error messages.
- You check the Spark UI and no jobs, stages, or tasks seem to run.

Potential treatments

After checking and confirming that the Spark UI environment tab shows the correct information for your application, it's worth double-checking your code. Many times, there might be a simple typo or incorrect column name that is preventing the Spark job from compiling into its underlying Spark plan (when using the DataFrame API).

- You should take a look at the error returned by Spark to confirm that there isn't an issue in your code, such as providing the wrong input file path or field name.
- Double-check to verify that the cluster has the network connectivity that you expect between your driver, your workers, and the storage system you are using.
- There might be issues with libraries or classpaths that are causing the wrong version of a library to be loaded for accessing storage. Try simplifying your application until you get a smaller version that reproduces the issue (e.g., just reading one dataset).

Errors During Execution

This kind of issue occurs when you already are working on a cluster or parts of your Spark Application run before you encounter an error. This can be a part of a scheduled job that runs at some interval or a part of some interactive exploration that seems to fail after some time.

Signs and symptoms

- One Spark job runs successfully on the entire cluster but the next one fails.
- A step in a multistep query fails.
- A scheduled job that ran yesterday is failing today.
- Difficult to parse error message.

Potential treatments

- Check to see if your data exists or is in the format that you expect. This can change over time or some upstream change may have had unintended consequences on your application.

- If an error quickly pops up when you run a query (i.e., before tasks are launched), it is most likely an *analysis error* while planning the query. This means that you likely misspelled a column name referenced in the query or that a column, view, or table you referenced does not exist.

- Read through the stack trace to try to find clues about what components are involved (e.g., what operator and stage it was running in).

- Try to isolate the issue by progressively double-checking input data and ensuring the data conforms to your expectations. Also try removing logic until you can isolate the problem in a smaller version of your application.

- If a job runs tasks for some time and then fails, it could be due to a problem with the input data itself, wherein the schema might be specified incorrectly or a particular row does not conform to the expected schema. For instance, sometimes your schema might specify that the data contains no nulls but your data does actually contain nulls, which can cause certain transformations to fail.

- It's also possible that your own code for processing the data is crashing, in which case Spark will show you the exception thrown by your code. In this case, you will see a task marked as "failed" on the Spark UI, and you can also view the logs on that machine to understand what it was doing when it failed. Try adding more logs inside your code to figure out which data record was being processed.

Slow Tasks or Stragglers

This issue is quite common when optimizing applications, and can occur either due to work not being evenly distributed across your machines ("skew"), or due to one of your machines being slower than the others (e.g., due to a hardware problem).

Signs and symptoms

Any of the following are appropriate symptoms of the issue:

- Spark stages seem to execute until there are only a handful of tasks left. Those tasks then take a long time.

- These slow tasks show up in the Spark UI and occur consistently on the same

- dataset(s). These occur in stages, one after the other.

- Scaling up the number of machines given to the Spark Application doesn't really help—some tasks still take much longer than others.

- In the Spark metrics, certain executors are reading and writing much more data than others.

Potential treatments

Slow tasks are often called "stragglers." There are many reasons they may occur, but most often the source of this issue is that your data is partitioned unevenly into DataFrame or RDD partitions. When this happens, some executors might need to work on much larger amounts of work than others. One particularly common case is that you use a group-by-key operation and one of the keys just has more data than others. In this case, when you look at the Spark UI, you might see that the shuffle data for some nodes is much larger than for others.

- Try increasing the number of partitions to have less data per partition.
- Try repartitioning by another combination of columns. For example, stragglers can come up when you partition by a skewed ID column, or a column where many values are null. In the latter case, it might make sense to first filter out the null values.
- Try increasing the memory allocated to your executors if possible.
- Monitor the executor that is having trouble and see if it is the same machine across jobs; you might also have an unhealthy executor or machine in your cluster—for example, one whose disk is nearly full.
- If this issue is associated with a join or an aggregation, see "Slow Joins" or "Slow Aggregations".
- Check whether your user-defined functions (UDFs) are wasteful in their object allocation or business logic. Try to convert them to DataFrame code if possible.
- Ensure that your UDFs or User-Defined Aggregate Functions (UDAFs) are running on a small enough batch of data. Oftentimes an aggregation can pull a lot of data into memory for a common key, leading to that executor having to do a lot more work than others.
- Turning on *speculation*, which we discuss in "Slow Reads and Writes", will have Spark run a second copy of tasks that are extremely slow. This can be helpful if the issue is due to a faulty node because the task will get to run on a faster one. Speculation does come at a cost, however, because it consumes additional resources. In addition, for some storage systems that use eventual consistency, you could end up with duplicate output data if your writes are not idempotent. (We discussed speculation configurations in Chapter 17.)
- Another common issue can arise when you're working with Datasets. Because Datasets perform a lot of object instantiation to convert records to Java objects for UDFs, they can cause a lot of garbage collection. If you're using Datasets, look at the garbage collection metrics in the Spark UI to see if they're consistent with the slow tasks.

Stragglers can be one of the most difficult issues to debug, simply because there are so many possible causes. However, in all likelihood, the cause will be some kind of data skew, so definitely begin by checking the Spark UI for imbalanced amounts of data across tasks.

Slow Aggregations

If you have a slow aggregation, start by reviewing the issues in the “Slow Tasks” section before proceeding. Having tried those, you might continue to see the same problem.

Signs and symptoms

- Slow tasks during a groupBy call.
- Jobs after the aggregation are slow, as well.

Potential treatments

Unfortunately, this issue can’t always be solved. Sometimes, the data in your job just has some skewed keys, and the operation you want to run on them needs to be slow.

- Increasing the number of partitions, prior to an aggregation, might help by reducing the number of different keys processed in each task.
- Increasing executor memory can help alleviate this issue, as well. If a single key has lots of data, this will allow its executor to spill to disk less often and finish faster, although it may still be much slower than executors processing other keys.
- If you find that tasks after the aggregation are also slow, this means that your dataset might have remained unbalanced after the aggregation. Try inserting a repart i t i on call to partition it randomly.
- Ensuring that all filters and SELECT statements that can be are above the aggregation can help to ensure that you’re working only on the data that you need to be working on and nothing else. Spark’s query optimizer will automatically do this for the structured APIs.
- Ensure null values are represented correctly (using Spark’s concept of nul l) and not as some default value like " " or "EMPTY". Spark often optimizes for skipping nulls early in the job when possible, but it can’t do so for your own placeholder values.
- Some aggregation functions are also just inherently slower than others. For instance, col l ect _l i st and col l ect _set are very slow aggregation functions because they *must* return all the matching objects to the driver, and should be avoided in performance-critical code.

Slow Joins

Joins and aggregations are both shuffles, so they share some of the same general symptoms as well as treatments.

Signs and symptoms

- A join stage seems to be taking a long time. This can be one task or many tasks.
- Stages before and after the join seem to be operating normally.

Potential treatments

- Many joins can be optimized (manually or automatically) to other types of joins. We covered how to select different join types in Chapter 8.
- Experimenting with different join orderings can really help speed up jobs, especially if some of those joins filter out a large amount of data; do those first.
- Partitioning a dataset prior to joining can be very helpful for reducing data movement across the cluster, especially if the same dataset will be used in multiple join operations. It's worth experimenting with different prejoin partitioning. Keep in mind, again, that this isn't "free" and does come at the cost of a shuffle.
- Slow joins can also be caused by data skew. There's not always a lot you can do here, but sizing up the Spark application and/or increasing the size of executors can help, as described in earlier sections.
- Ensuring that all filters and select statements that can be are above the join can help to ensure that you're working only on the data that you need for the join.
- Ensure that null values are handled correctly (that you're using nul l) and not some default value like " " or "EMPTY", as with aggregations.
- Sometimes Spark can't properly plan for a broadcast join if it doesn't know any statistics about the input DataFrame or table. If you know that one of the tables that you are joining is small, you can try to force a broadcast (as discussed in Chapter 8), or use Spark's statistics collection commands to let it analyze the table.

Slow Reads and Writes

Slow I/O can be difficult to diagnose, especially with networked file systems.

Signs and symptoms

- Slow reading of data from a distributed file system or external system.
- Slow writes from network file systems or blob storage.

Potential treatments

- Turning on speculation (set spark. specul at i on to t rue) can help with slow reads and writes. This will launch additional tasks with the same operation in an attempt to see whether it's just some transient issue in the first task. Speculation is a powerful tool and works well with consistent file systems. However, it can cause duplicate data writes with some eventually consistent cloud services, such as Amazon S3, so check whether it is supported by the storage system connector you are using.
- Ensuring sufficient network connectivity can be important—your Spark cluster may simply not have enough total network bandwidth to get to your storage system.
- For distributed file systems such as HDFS running on the same nodes as Spark, make sure Spark sees the same hostnames for nodes as the file system. This will enable Spark to do locality-aware scheduling, which you will be able to see in the "locality" column in the Spark UI. We'll talk about locality a bit more in the next chapter.

Driver OutOfMemoryError or Driver Unresponsive

This is usually a pretty serious issue because it will crash your Spark Application. It often happens due to collecting too much data back to the driver, making it run out of memory.

Signs and symptoms

- Spark Application is unresponsive or crashed.
- Out Of MemoryErrors or garbage collection messages in the driver logs.
- Commands take a very long time to run or don't run at all.
- Interactivity is very low or non-existent.
- Memory usage is high for the driver JVM.

Potential treatments

There are a variety of potential reasons for this happening, and diagnosis is not always straightforward.

- Your code might have tried to collect an overly large dataset to the driver node using operations such as col l ect .
- You might be using a broadcast join where the data to be broadcast is too big. Use Spark's maximum broadcast join configuration to better control the size it will broadcast.

- A long-running application generated a large number of objects on the driver and is unable to release them. Java's *jmap* tool can be useful to see what objects are filling most of the memory of your driver JVM by printing a histogram of the heap. However, take note that *jmap* will pause that JVM while running.
- Increase the driver's memory allocation if possible to let it work with more data.
- Issues with JVMs running out of memory can happen if you are using another language binding, such as Python, due to data conversion between the two requiring too much memory in the JVM. Try to see whether your issue is specific to your chosen language and bring back less data to the driver node, or write it to a file instead of bringing it back as in-memory objects.
- If you are sharing a SparkContext with other users (e.g., through the SQL JDBC server and some notebook environments), ensure that people aren't trying to do something that might be causing large amounts of memory allocation in the driver (like working overly large arrays in their code or collecting large datasets).

Executor OutOfMemoryError or Executor Unresponsive

Spark applications can sometimes recover from this automatically, depending on the true underlying issue.

Signs and symptoms

- Out Of MemoryErrors or garbage collection messages in the executor logs. You can find these in the Spark UI.
- Executors that crash or become unresponsive.
- Slow tasks on certain nodes that never seem to recover.

Potential treatments

- Try increasing the memory available to executors and the number of executors.
- Try increasing PySpark worker size via the relevant Python configurations.
- Look for garbage collection error messages in the executor logs. Some of the tasks that are running, especially if you're using UDFs, can be creating lots of objects that need to be garbage collected. Repartition your data to increase parallelism, reduce the amount of records per task, and ensure that all executors are getting the same amount of work.
- Ensure that null values are handled correctly (that you're using nul l) and not some default value like " " or "EMPTY", as we discussed earlier.

- This is more likely to happen with RDDs or with Datasets because of object instantiations. Try using fewer UDFs and more of Spark's structured operations when possible.
- Use Java monitoring tools such as *jmap* to get a histogram of heap memory usage on your executors, and see which classes are taking up the most space.
- If executors are being placed on nodes that also have other workloads running on them, such as a key-value store, try to isolate your Spark jobs from other jobs.

Unexpected Nulls in Results

Signs and symptoms

- Unexpected nul l values after transformations.
- Scheduled production jobs that used to work no longer work, or no longer produce the right results.

Potential treatments

- It's possible that your data format has changed without adjusting your business logic. This means that code that worked before is no longer valid.
- Use an accumulator to try to count records or certain types, as well as parsing or processing errors where you skip a record. This can be helpful because you might think that you're parsing data of a certain format, but some of the data doesn't. Most often, users will place the accumulator in a UDF when they are parsing their raw data into a more controlled format and perform the counts there. This allows you to count valid and invalid records and then operate accordingly after the fact.
- Ensure that your transformations actually result in valid query plans. Spark SQL sometimes does implicit type coercions that can cause confusing results. For instance, the SQL expression SELECT 5*"23" results in 115 because the string "25" converts to an the value 25 as an integer, but the expression SELECT 5 * " " results in nul l because casting the empty string to an integer gives nul l . Make sure that your intermediate datasets have the schema you expect them to (try using pri nt Schema on them), and look for any CAST operations in the final query plan.

No Space Left on Disk Errors

Signs and symptoms

- You see "no space left on disk" errors and your jobs fail.