A Gentle Introduction to

databricks

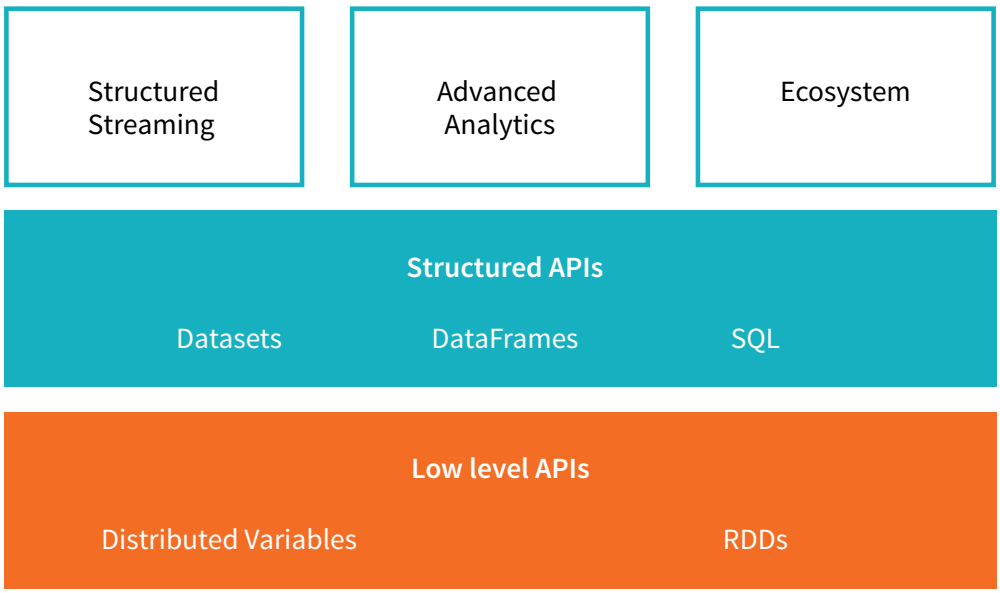
Preface

Apache Spark has seen immense growth over the past several years. The size and scale of Spark Summit 2017 is a true reflection of innovation after innovation that has made itself into the Apache Spark project. Databricks is proud to share excerpts from the upcoming book, Spark: The Definitive Guide. Enjoy this free preview copy, courtesy of Databricks.

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 the time this writing, Spark is the most actively developed open source engine for this task; making it the de facto 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.

Here's a simple illustration of all that Spark has to offer an end user.



You'll notice the boxes 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 opened this book, you may already know a little bit about Apache Spark and what it can do. Nonetheless, in this chapter, we want to cover a bit about 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 cover 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.

1. 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 *API*s. 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 can be used to build an application out of smaller pieces or out of existing libraries, and 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 high-performance 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 built-in 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 poweful optimization under user applications.

2. Computing Engine: At the same time that Spark strives for unification, Spark carefully limits its scope to a *computing engine*. By this, we mean that Spark only handles loading data from storage systems and performing computation on it, not permanent storage as the end itself. Spark can be used 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 of these. 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 user-facing APIs, Spark works hard to make these storage systems look largely similar so that applications do not have 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 hard to run one of the systems without the other, or even more importantly, to write applications that access data stored anywhere else. While Spark runs well on Hadoop storage, it is also now used broadly in environments where the Hadoop architecture does not make sense, such as the public cloud (where storage can be purchased separately from computing) or streaming applications.

3. 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, and 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 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 computer programming, this is due to changes in the economic trends that underlie computer applications and hardware.

For most of their history, computers got faster every year through processor speed increases: the new processors each year could run more instructions per second than last year's. As a result, applications also automatically got faster every year, without having to change their code. This trend led to a large and established ecosystem of applications building up over time, most of which were only designed to run on a single processor, and rode the trend of improved processor speeds to scale up to larger computations or 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 towards adding more parallel CPU cores all running at the same speed. This change meant that, all of a sudden, applications needed to be modified to add parallelism in order to run faster, and already started to 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 2x 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 only costs 3-4 US dollars; 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 where collecting data is extremely inexpensive - many organizations might 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 created for.

History of Spark

Apache Spark began in 2009 as the Spark research project at UC Berkeley, which was first published in a research paper in 2010 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 start 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 started 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 multi-step applications, and then implemented it over a new engine that could perform efficient, in-memory data sharing across computation steps. They also began testing this system with both Berkeley and external users.

The first version of Spark only supported 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 started to follow the “standard library” approach it has today. In particular, different AMPlab groups started MLlib (Apache Spark’s machine learning library), Spark Streaming, and GraphX (a graph processing API). 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 over 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 startup 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. Starting in 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 Day and Future of Spark

Spark has been around for a number of years at this point but continues to gain massive popularity. Many new projects within the Spark ecosystem continue to push the boundaries of what's possible with the system. For instance a streaming engine (Structured Streaming) was built and introduced into Spark in 2017 and 2017. This technology is a huge apart of companies solving massive scale data challenges, from technology companies like Uber and Netflix leveraging Spark's streaming engines and machine learning tools to institutions like the Broad Institute of MIT and Harvard leveraging Spark for genetic data analysis.

Spark will continue to be a cornerstone of companies doing big data analysis for the foreseeable future, especially since Spark is still in its infancy. Any data scientist or data engineer that needs to solve big data problems needs a copy of Spark on their machine and a copy of this book on their book shelf!

Running Spark

This book contains an abundance of Spark related code. Therefore, as a reader, it is essential that you're prepared to run the code in this book. For the most part, you'll want to run the code interactively so that you can experiment with the code. Let's go over some of your options before we get started with the coding parts of the book.

Spark can be used from Python, Java, or Scala, R, or SQL. Spark itself is written in Scala, and runs on the Java Virtual Machine (JVM) and therefore to run Spark either on your laptop or a cluster, all you need is an installation of Java 6 or newer. If you wish to use the Python API you will also need a Python interpreter (version 2.6 or newer). If you wish to use R you'll also need a version of R on your machine.

There are two versions we'll recommend using to get started with Spark, you can either download it or run it for free on the web on Databricks Community Edition. We explain both of those options below.

Downloading Spark

The first step to using Spark is to download and unpack it. Let's start by downloading a recent precompiled released version of Spark. Visit http://spark.apache.org/downloads.html, select the package type of "Pre-built for Hadoop 2.7 and later" and click "Direct Download." This will download a compressed TAR file, or tarball. The majority of this book was written during the release of Spark 2.1 and 2.2 so downloading any version 2.2 or greater should be a good starting point.

Downloading Spark for your Hadoop Version

You don't need to have Hadoop, but if you have an existing Hadoop cluster or HDFS installation, download the matching version. You can do so from http://spark.apache.org/downloads.html by selecting a different package type, but they will have slightly different filenames. We discuss how Spark runs on clusters in later chapters, but at this point we recommend running a pre-compiled binary on your local machine to start out.

NOTE: In Spark 2.2, Spark maintainers added the ability to install Spark via pip. This is brand new at the time of this writing. This will likely be the best way to install Pyspark in the future but because it's a brand new, the community has not had a chance to test it thoroughly across a multitude of machines and environments.

Building Spark from Source

We won't cover this in the book but you can also build Spark from source. You can find the latest source code on GitHub or select the package type of "Source Code" when downloading.

Once you've downloaded Spark, you'll want to open up a command line prompt and unzip the package. In our case, we're unzipping Spark 2.1. The following is a code snippet that you can run on any unix style command line in order to unzip the file you downloaded from Spark and move into the directory.

```
cd ~/Downloads

tar -xf spark-2.1.0-bin-hadoop2.7.tgz

cd spark-2.1.0-bin-hadoop2.7.tgz
```

Now Spark has a large number of directories and files within the project. Don't be intimidated! We will cover the important ones in the book but for right now. Spark also has a large number of deployment modes, we will cover those at the end of the book. You will be able to run everything in this book on your local machine. So only jump to those when necessary.

Launching Spark's Consoles

The first step you should take is launching Spark and this all depends on our language familiarity. The majority of this book is written with Scala, Python, and SQL in mind and those are our recommended starting points.

Launching the Python Console

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

```
./bin/pyspark
```

Once you've done that, type `spark` and hit enter. You'll see the SparkSession (which we will cover in the next chapter.)

Launching the Scala Console

In order to launch the scala console, you'll need to have some java runtime installed on your machine.

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

Once you've done that, type `spark` and hit enter. You'll see the SparkSession (which we will cover in the next chapter.)

Launching the SQL Console

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

```
./bin/spark-sql
```

Running Spark in the Cloud

Now for the more adventurous, it's worth running Spark on your local machine. For those that would rather have a clean, interactive notebook experience with Spark. It's worth using Databricks' Community Version. Databricks, as we mentioned above, is the company founded by the authors of Apache Spark and in an effort to continue to serve the Spark community, they released a free community edition that anyone can use to run Spark without the overhead and management of managing your own Spark cluster. Additionally, Databricks makes all of the data used in this book accessible from the Community Edition so that it's readily accessible. You can access and import all the code from this book into Community Edition by following these instructions in the github repository for this book to get started.

Data used in this book

We'll use a number of data sources in this book for our examples. You can download them from the official repository in this book by following these instructions on the github repository for this book. In short, you'll download the data, put it in a folder, and then run the code snippets in this book!

A Gentle Introduction to Spark

Now that we took our history lesson on Apache Spark, it's time to start using it and applying it! This chapter will present a gentle introduction to Spark - 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 are empowered start using Spark right away. Let's get started with some basic background terminology and concepts.

Spark's Basic Architecture

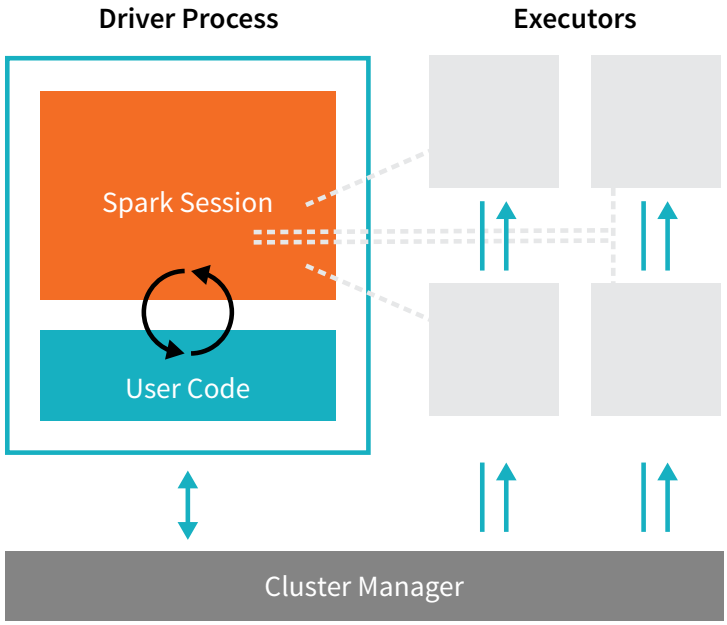
Typically when you think of a "computer" you think about one machine sitting on your desk at home or at work. This machine works perfectly well for watching movies or working with spreadsheet software. However, as many users likely experience at some point, there are some things that your computer is 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 may not have time to wait for the computation to finish). A *cluster*, or group of machines, pools the resources of many machines together allowing us to use all the cumulative resources as if they were one. Now a group of machines alone is not powerful, you need a framework to coordinate work across them. Spark is a tool for just that, managing and coordinating the execution of tasks on data across a cluster of computers.

The cluster of machines that Spark will leverage to execute tasks will be 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 `main()` 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 (defined 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 executing the work that the driver assigns them. This means, 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.



The cluster manager controls physical machines and allocates resources to Spark Applications. This can be one of several 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 talk more in depth about cluster managers in Part IV: Production Applications of this book.

In the previous illustration we see on the left, our driver and on the right the 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, this means that they can live on the same machine or different machines. In local mode, these both run (as threads) on your individual computer instead of a cluster. We wrote this book with local mode in mind, so everything should be runnable on a single machine.

As a short review of Spark Applications, the key points to understand at this point are that:

- Spark has some cluster manager that maintains an understanding of the resources available.
- The driver process is responsible for executing our driver program's commands accross the executors in order to complete our task.

Now while our executors, for the most part, will always be running Spark code. Our driver can be "driven" from a number of different languages through Spark's Language APIs.

Spark's Language APIs

Spark's language APIs allow you to run Spark code from other langauges. For the most part, Spark presents some core "concepts" in every language and these concepts are translated into Spark code that runs on the cluster of machines. If you use the Structured APIs (Part II of this book), you can expect all languages to have the same performance characteristics.

NOTE

> This is a bit more nuanced than we are letting on at this point but for now, it's the right amount of information for new users. In Part II of this book, we'll dive into the details of how this actually works.

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.

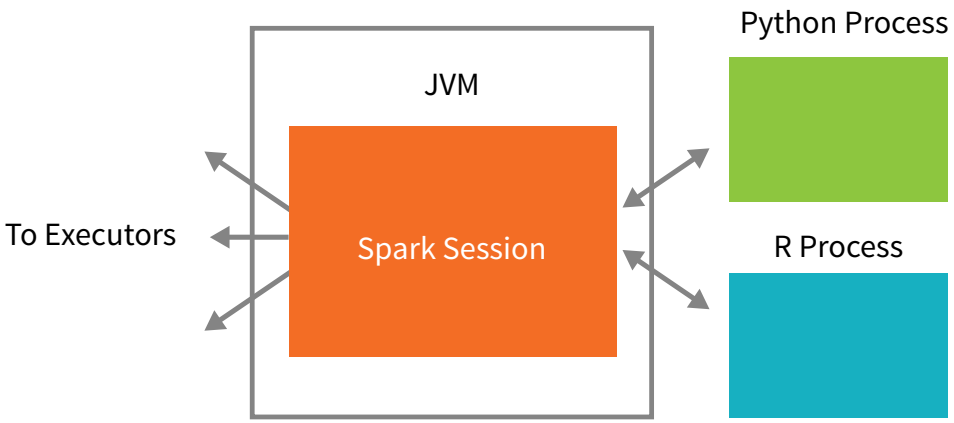
SQL

Spark supports ANSI SQL 2003 standard. This makes it easy for analysts and non-programmers to leverage the big data powers of Spark. This book will include SQL code examples wherever relevant

R

Spark has two commonly used R libraries, one as a part of Spark core (SparkR) and another as a R community driven package (sparklyr). We will cover these two different integrations in Part VII: Ecosystem.

Here's a simple illustration of this relationship.



Each language API will maintain the same core concepts that we described above. There is a SparkSession available to the user, the SparkSession will be the entrance point to running Spark code. When using Spark from a Python or R, the user never writes explicit JVM instructions, but instead writes Python and R code that Spark will translate into code that Spark can then run on the executor JVMs.

Spark's APIs

While Spark is available from a variety of languages, what Spark 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 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 the Spark Application. We do that with a SparkSession.

NOTE
To do this we will start Spark's local mode, just like we did in the previous chapter. This means running `./bin/spark-shell` to access the Scala console to start an interactive session. You can also start Python console with `./bin/pyspark`. This starts an interactive Spark Application. There is also a process for submitting standalone applications to Spark called `spark-submit` where you can submit a precompiled application to Spark. We'll show you how to do that in the next chapter.

When we start Spark in this interactive mode, we implicitly create a SparkSession which manages the Spark Application. When we start it through a job submission, we must go about creating it or accessing it.

The SparkSession

As discussed in the beginning of this chapter, we control our Spark Application through a driver process. This driver process manifests itself to the user as an object called the SparkSession. The SparkSession instance is the way Spark executes user-defined manipulations across the cluster. There is a one to one correspondance between a SparkSession and a Spark Application. In Scala and Python the variable is available as `spark` when you start up the console. Let's go ahead and look at the `SparkSession` in both Scala and/or Python.

```
spark
```

In Scala, you should see something like:

```
res0: org.apache.spark.sql.SparkSession = org.apache.spark.sql.SparkSession@27159a24
```

In Python you'll see something like:

```
<pyspark.sql.session.SparkSession at 0x7efda4c1ccd0>
```

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.

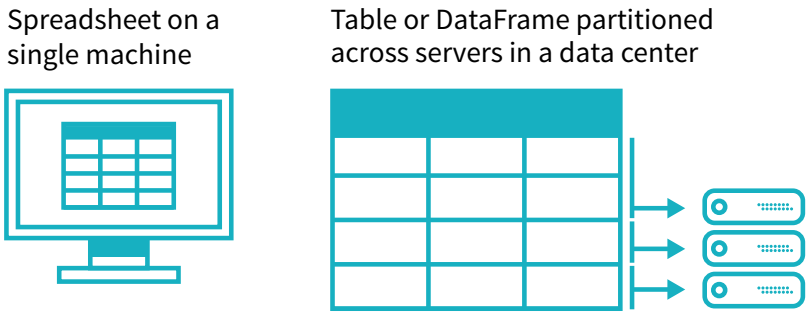
```
%scala

val myRange = spark.range(1000).toDF("number")
```

```
%python

myRange = spark.range(1000).toDF("number")
```

You just ran your first Spark code! We created a DataFrame with one column containing 1000 rows with values from 0 to 999. This range of number 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 of columns and the types in those columns the *schema*. A simple analogy would be a spreadsheet with named columns. The fundamental difference is that while a spreadsheet sits on one computer in one specific location, 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.



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 in python and R to the resources that exist on that specific machine. However, since Spark has language interfaces for both Python and R, it's quite easy to convert to Pandas (Python) DataFrames to Spark DataFrames and R DataFrames to Spark DataFrames (in R).

NOTE
Spark has several core abstractions: Datasets, DataFrames, SQL Tables, and Resilient Distributed Datasets (RDDs). These abstractions all represent distributed collections of data however they have different interfaces for working with that 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 of this book. The following concepts apply to all of the core abstractions.

Partitions

In order 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 our cluster. A DataFrame's partitions represent how the data is physically distributed across your cluster of machines during execution. If you have one partition, Spark will only have a parallelism of one even if you have thousands of executors. If you have many partitions, but only one executor Spark will still only have a parallelism of one because there is only one computation resource.

An important thing to note, is that with DataFrames, we do not (for the most part) manipulate partitions manually (on an individual basis). We 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 Resilient Distributed Datasets interface) and we cover those in Part III of this book.

Transformations

In Spark, the core data structures are *immutable* meaning they cannot be changed once created. This might seem like a strange concept at first, if you cannot change it, how are you supposed to use it? In order to "change" a DataFrame you will have to instruct Spark how you would like to modify the DataFrame you have into the one that you want. These instructions are called *transformations*. Let's perform a simple transformation to find all even numbers in our currentDataFrame.

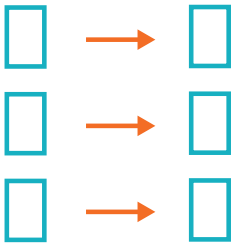
```
%scala

val divisBy2 = myRange.where("number % 2 = 0")
```

```
%python

divisBy2 = myRange.where("number % 2 = 0")
```

You will notice that these return no output, that's because we only specified an abstract transformation and Spark will not act on transformations until we call an action, discussed shortly. Transformations are the core of how you will be expressing your business logic using Spark. There are two types of transformations, those that specify narrow dependencies and those that specify wide dependencies.

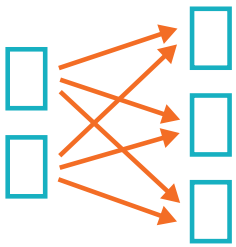
Transformations consisting of *narrow dependencies* (we'll call them narrow transformations) are those where each input partition will contribute to only one output partition. In the preceding code snippet, our `where` statement specifies a narrow dependency, where only one partition contributes to at most one output partition.

Narrow Transformations
1 to 1



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* where Spark will exchange partitions across the cluster. With narrow transformations, Spark will automatically perform an operation called pipelining on narrow dependencies, this means 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 will write the results to disk. You'll see lots of talks about shuffle optimization across the web because it's an important topic but for now all you need to understand are that there are two kinds of transformations.

Wide Transformations (shuffles)
1 to N



We now 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 we express some operation, we build up a *plan* of transformations that we would like to apply to our source data. Spark, by waiting until the last minute to execute the code, will compile this plan from your raw, DataFrame transformations, to an efficient physical plan that will run as efficiently as possible across the cluster. This provides immense benefits to the end user 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()
```

We now see a result! There are 500 number divisible by two from o to 999 (big surprise!). Now `count` is not the only action. There are three kinds of actions:

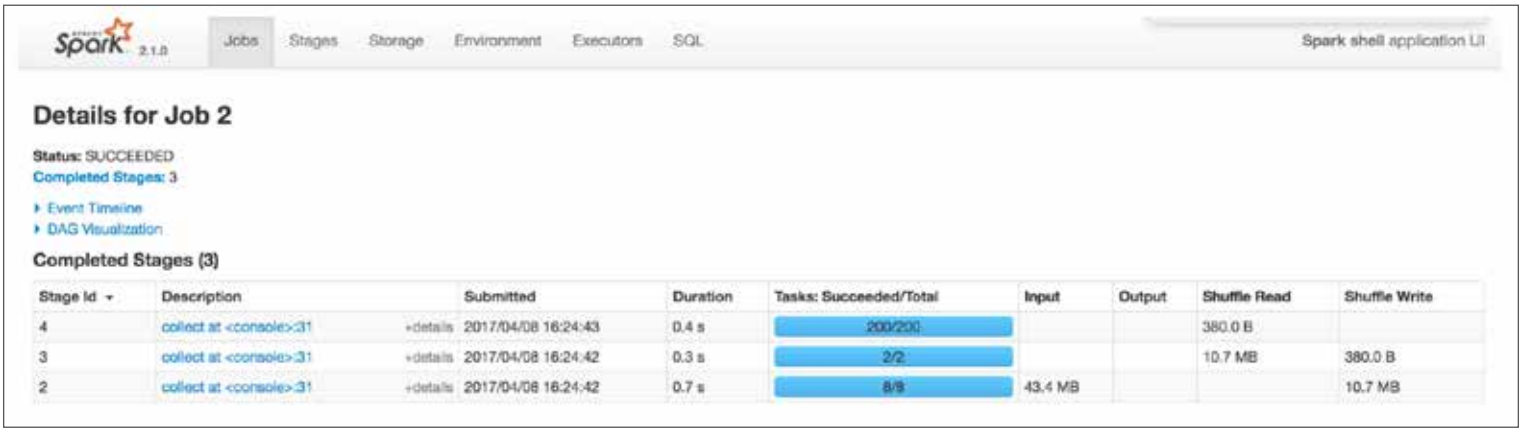
- actions to view data in the console;
- actions to collect data to native objects in the respective language;
- and actions to write to output data sources.

In specifying our 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, then a collect with brings our result to a native object in the respective language. We can see all of this by inspecting the Spark UI, a tool included in Spark that allows us to monitor the Spark jobs running on a cluster.

Spark UI

During Spark's execution of the previous code block, users can monitor the progress of their job through the Spark UI. The Spark UI is available on port 4040 of the driver node. If you are running in local mode this will just be the `http://localhost:4040`. The Spark UI maintains information on the state of our Spark jobs, environment, and

cluster state. It's very useful, especially for tuning and debugging. In this case, we can see one Spark job with two stages and nine tasks were executed.



This chapter avoids the details of Spark jobs and the Spark UI, we cover the Spark UI in detail in Part IV: Production Applications. At this point you should understand that a Spark job represents a set of transformations triggered by an individual action and we can monitor that 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 worked example and explaining step by step what is happening under the hood. We'll be using some flight data available here from the United States Bureau of Transportation statistics.

Inside of the CSV folder linked above, you'll see that we have a number of files. You will also notice a number of other folders with different file formats that we will discuss in Part II: Reading and Writing data. We will focus on the CSV files.

Each file has a number of rows inside of it. Now these files are CSV files, meaning that they're a semi-structured data format with a row in the file representing a row in our future DataFrame.

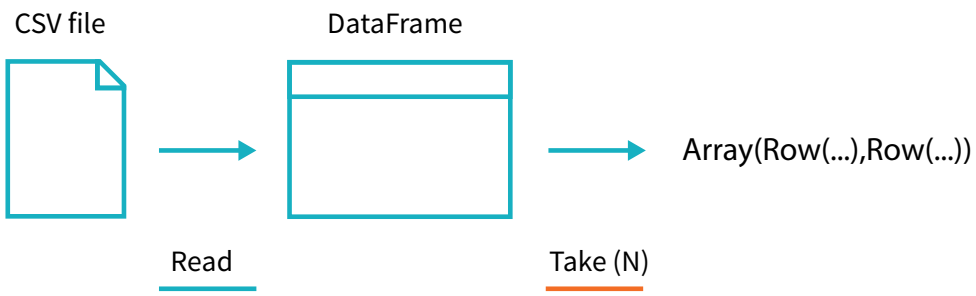
```
$ head /mnt/defg/flight-data/csv/2015-summary.csv

DEST_COUNTRY_NAME,ORIGIN_COUNTRY_NAME,count

United States,Romania,15

United States,Croatia,1

United States,Ireland,344
```

Spark includes the ability to read and write from a large number of data sources. In order to read this data in, 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, we want Spark to take a best guess at what the schema of our DataFrame should be. The reason for this is that CSV files are not completely structured data formats. We also want to specify that the first row is the header in the file, we'll specify that as an option too.

To get this information Spark will read in a little bit of the data and then attempt to parse the types in those rows according to the types available in Spark. You'll see that this works just fine. We also have the option of strictly specifying a schema when we read in data (which we recommend in production scenarios).

```
%scala

val flightData2015 = spark

    .read
    .option("inferSchema", "true")
    .option("header", "true")
    .csv("/mnt/defg/flight-data/csv/2015-summary.csv")
```

```
%python

flightData2015 = spark\

    .read\
    .option("inferSchema", "true")\
    .option("header", "true")\
    .csv("/mnt/defg/flight-data/csv/2015-summary.csv")
```



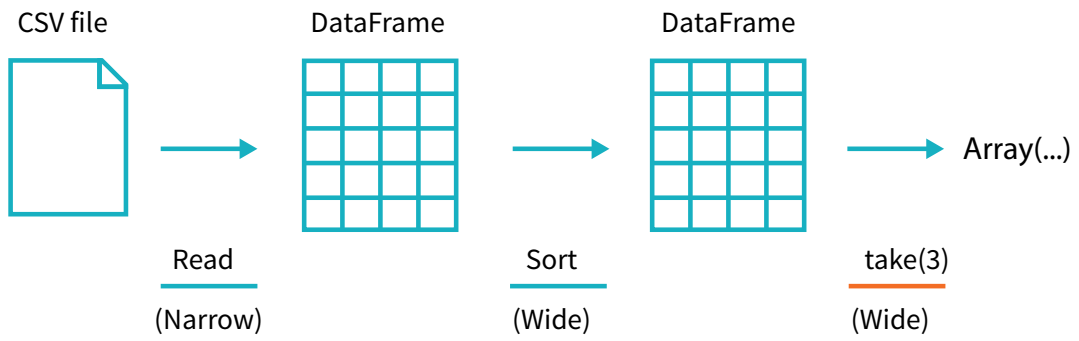
Each of these DataFrames (in Scala and Python) each have a set of columns with an unspecified number of rows. The reason the number of rows is "unspecified" is because reading data is a transformation, and is therefore a lazy operation. Spark only peeked at a couple of rows of data to try to guess what types each column should be.

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 we will sort our data according to the count column which is an integer type.

note
Remember, the `sort` does not modify the DataFrame. We use the sort is a transformation that returns a new DataFrame by transforming the previous DataFrame. Let's illustrate what's happening when we call `take` on that resulting DataFrame.



Nothing hpapens 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 `explain` plan. We can call explain on any DataFrame object to see the DataFrame's lineage (or how Spark will execute this query).

```
flightData2015.sort("count").explain()
```

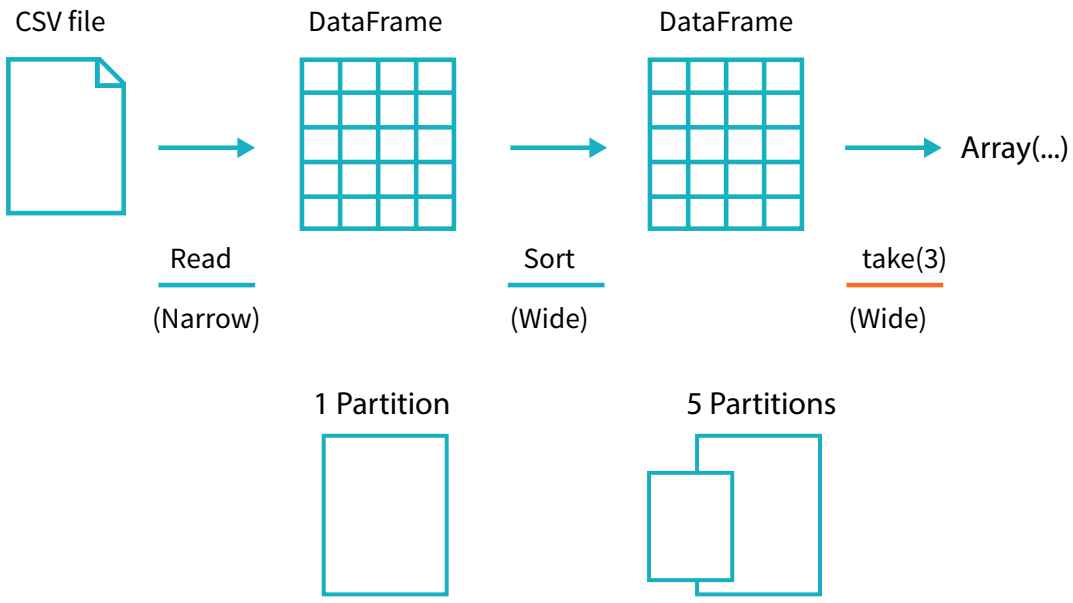
Congratulations, you've just read your first explain plan! Explain plans are a bit arcane, but with a bit of practice it becomes second nature. Explain plans can be read from top to bottom, the top being the end result and the

bottom being the source(s) of data. In our case, just 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 have 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 in order to kick off this plan. However before doing that, we’re going to set a configuration. By default, when we perform a shuffle Spark will output two hundred shuffle partitions. We will set this value to five in order to reduce the number of the output partitions from the shuffle from two hundred to five.

```
spark.conf.set(“spark.sql.shuffle.partitions”, “5”)

flightData2015.sort(“count”).take(2)

... Array([United States,Singapore,1], [Moldova,United States,1])
```

This operation is illustrated in the following image. You’ll notice that in addition to the logical transformations, we include the physical partition count as well.



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, but rather configure physical execution characteristics through things like the shuffle partitions parameter we set above. We got five output partitions because that's what we changed the shuffle partition value to. 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 run times. Remeber 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 our jobs.

DataFrames and SQL

We worked through a simple example in the previous example, let's now work through a more complex example and follow along in both DataFrames and SQL. Spark 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 we see in the explain plan) before actually executing your code. Spark SQL allows you as a user to 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.

Any DataFrame can be made into a table or view with one simple method call.

```
%scala

flightData2015.createOrReplaceTempView("flight_data_2015")
```

```
%python

flightData2015.createOrReplaceTempView("flight_data_2015")
```

Now we can query our data in SQL. To execute a SQL query, we'll use the `spark.sql` function (remember `spark` is our SparkSession variable?) that conveniently, returns a new DataFrame. While this may seem a bit circular in logic - that a SQL query against a DataFrame returns another DataFrame, it's actually quite powerful. As a user, you can specify transformations in the manner most convenient to you at any given point in time and not have to trade any efficiency to do so! To understand that this is happening, let's take a look at two explain plans.

```
%scala

val sqlWay = spark.sql("""

SELECT DEST_COUNTRY_NAME, count(1)
FROM flight_data_2015
GROUP BY DEST_COUNTRY_NAME
""")

val dataFrameWay = flightData2015
    .groupBy('DEST_COUNTRY_NAME)
    .count()

sqlWay.explain
dataFrameWay.explain
```

```
%python

sqlWay = spark.sql("""
SELECT DEST_COUNTRY_NAME, count(1)
FROM flight_data_2015
GROUP BY DEST_COUNTRY_NAME
""")

dataFrameWay = flightData2015\
.groupBy("DEST_COUNTRY_NAME")\
.count()

sqlWay.explain()
dataFrameWay.explain()
```

```
== 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] ...
== 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] ...
```

We can see that these plans compile to the exact same underlying plan!

To reinforce the tools available to us, 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 leverage and import to help you resolve your big data problems faster. We will use the `max` function, to find out what the maximum number of flights to and from any given location are. This just scans each value in relevant column the DataFrame and sees if it's bigger than the previous values that have been seen. This is a transformation, as 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)
```

```
%scala

import org.apache.spark.sql.functions.max

flightData2015.select(max("count")).take(1)
```

```
%python

from pyspark.sql.functions import max

flightData2015.select(max("count")).take(1)
```

Great, that's a simple example. Let's perform something a bit more complicated and find out the top five destination countries in the data? This is a our first multi-transformation query so we'll take it step by step. We will start with a fairly straightforward SQL aggregation.

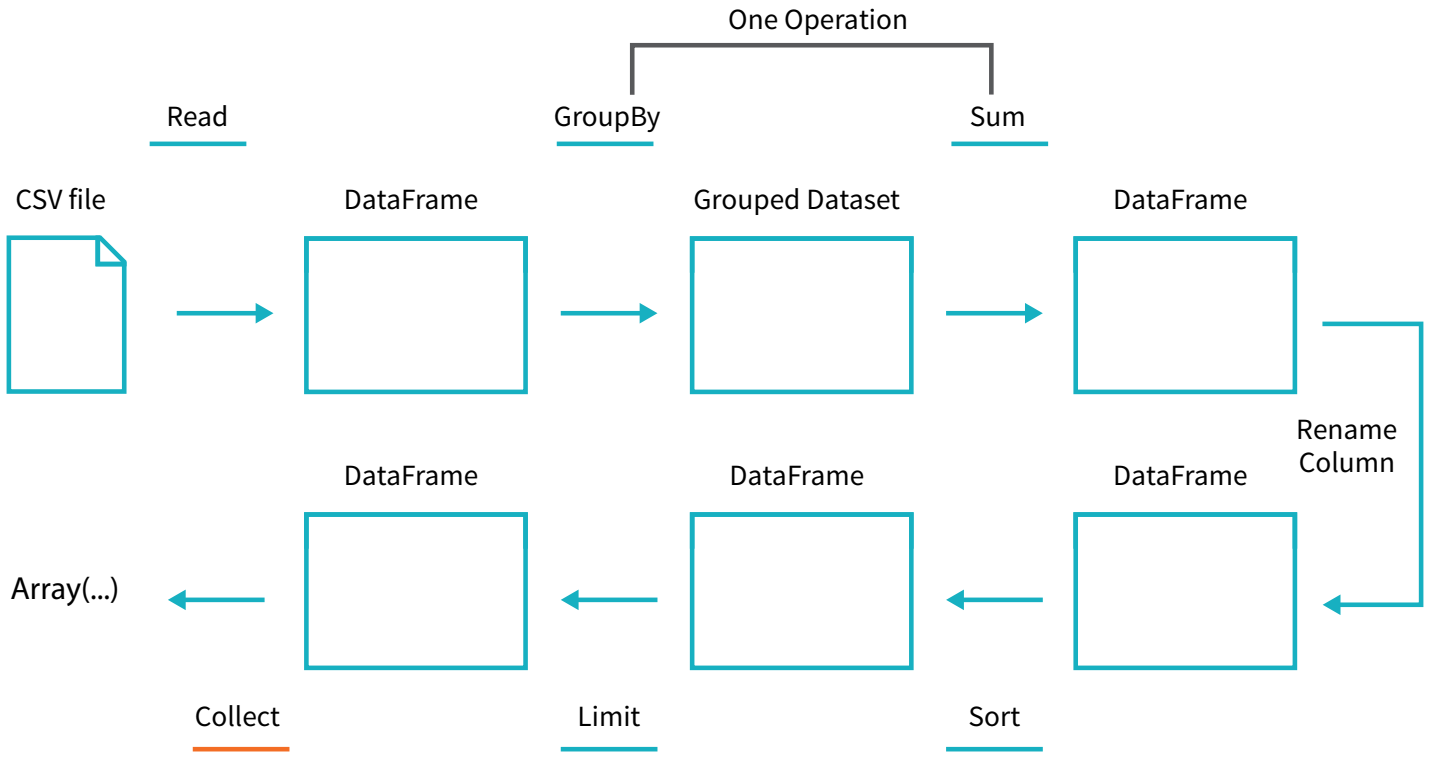
```
%scala

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.collect()
```

```
%python

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.collect()
```

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 execute the queries and see their results as a sanity check.

```
%scala

import org.apache.spark.sql.functions.desc

flightData2015
    .groupBy("DEST_COUNTRY_NAME")
    .sum("count")
    .withColumnRenamed("sum(count)", "destination_total")
    .sort(desc("destination_total"))
    .limit(5)
    .collect()
```

```
%python

from pyspark.sql.functions import desc

flightData2015\
    .groupBy("DEST_COUNTRY_NAME")\
    .sum("count")\
    .withColumnRenamed("sum(count)", "destination_total")\
    .sort(desc("destination_total"))\
    .limit(5)\
    .collect()
```



Now there are 7 steps that take us all the way back to the source data. You can see this in the explain plan on those DataFrames. Illustrated below are the set of steps that we perform in "code". The true execution plan (the one visible in explain) will differ from what we have below because of optimizations in 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.

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 RelationalGroupedDataset 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 can see this by trying to perform an action on it (which will not work). 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 the type information we have supplied.

The fourth step is a simple renaming, we use the `withColumnRenamed` 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 be the largest values found in the `destination_total` column.

You likely noticed that we had to import a function to do this, the `desc` function. You might also notice that `desc` does not return a string but a `Column`. In general, many DataFrame methods will accept Strings (as column names) or `Column` 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 five values. This is just like a filter except that it filters by position instead of by value. It's safe to say that it basically just specifies a `DataFrame` of a certain size.

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

```
%scala

flightData2015
    .groupBy("DEST_COUNTRY_NAME")
    .sum("count")
    .withColumnRenamed("sum(count)", "destination_total")
    .sort(desc("destination_total"))
    .limit(5)
    .explain()
```

```
%python

flightData2015\

.groupBy("DEST_COUNTRY_NAME")\
.sum("count")\
.withColumnRenamed("sum(count)", "destination_total")\
.sort(desc("destination_total"))\
    .limit(5)\
    .explain()
```

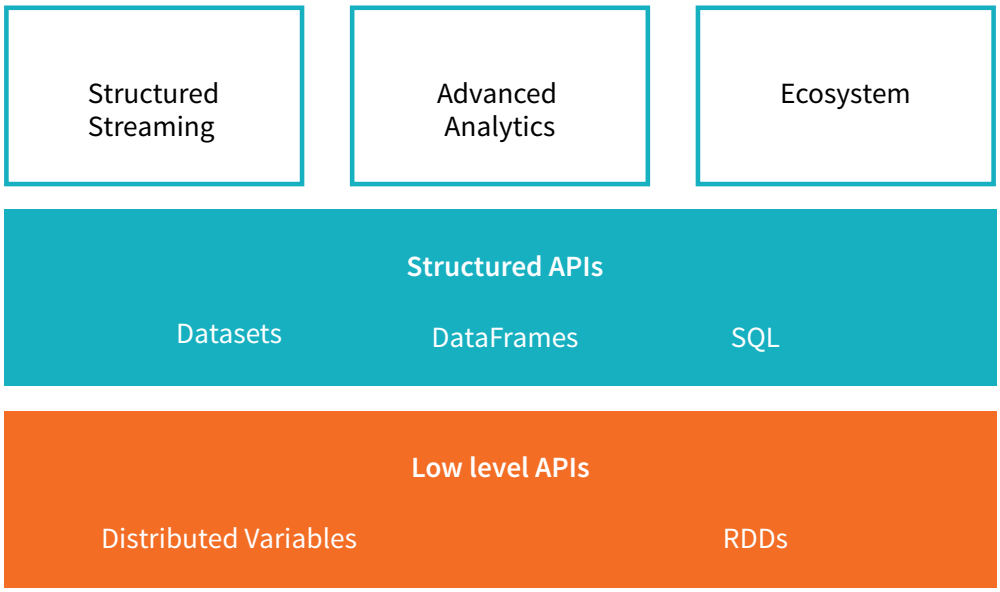
```
== Physical Plan ==
TakeOrderedAndProject(limit=5, orderBy=[destination_total#16194L DESC], output=[DEST_COUNTRY_NAME#7323,...

+- *HashAggregate(keys=[DEST_COUNTRY_NAME#7323], functions=[sum(count#7325L)])
  +- Exchange hashpartitioning(DEST_COUNTRY_NAME#7323, 5)
      +- *HashAggregate(keys=[DEST_COUNTRY_NAME#7323], functions=[partial
sum(count#7325L)])
         +- InMemoryTableScan [DEST_COUNTRY_NAME#7323, count#7325L]
               +- InMemoryRelation [DEST_COUNTRY_NAME#7323, ORIGIN_COUNTRY_NAME#7324, count#7325L]...
                     +- *Scan csv [DEST_COUNTRY_NAME#7578,ORIGIN_COUNTRY_NAME#7579,count#7580L]...
```

While this explain plan doesn't match our exact "conceptual plan" all of the pieces are there. You can see the limit statement as well as the `orderBy` (in the first line). You can also see how our aggregation happens in two phases, in the `partial_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 have to collect the data. We can also write it out to any data source that Spark supports. For instance, let's say that we wanted to store the information in a database like PostgreSQL or write them out to another file.

A Tour of Spark's Toolset

In the previous chapter 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. Spark is composed of the simple primitives, the lower level APIs and the Structured APIs, then a series of "standard libraries" included in Spark.



Developers use these tools for a variety of different tasks, from graph analysis and machine learning to streaming and integrations with a host of libraries and databases. This chapter will present a whirlwind tour of much of what Spark has to offer. Each section in this chapter are elaborated upon by other parts of this book, this chapter is simply here to show you what's possible.

This chapter will cover:

- Production applications with spark-submit,
- Datasets: structured and type safe APIs,
- Structured Streaming,
- Machine learning and advanced analytics,

- Spark's lower level APIs,
- SparkR,
- Spark's package ecosystem.

The entire book covers these topics in depth, the goal of this chapter is simply to provide a whirlwind tour of Spark. Once you've gotten the tour, you'll be able to jump to many different parts of the book to find answers to your questions about particular topics. This chapter aims for breadth, instead of depth. Let's get started!

Production Applications

Spark makes it easy to make simple to reason about and simple to evolve big data programs. Spark also makes it easy to turn in your interactive exploration into production applications with a tool called `spark-submit` that is included in the core of Spark. spark-submit does one thing, it allows you to submit your applications to a currently managed cluster to run. When you submit this, the application will run until the application exists or errors. You can do this with all of Spark's support cluster managers including Standalone, Mesos, and YARN.

In the process of doing so, you have a number of knobs that you can turn and control to specify the resources this application has as well, how it should be run, and the parameters for your specific application.

You can write these production applications in any of Spark's supported languages and then submit those applications for execution. The simplest example is one that you can do on your local machine by running the following command line snippet on your local machine in the directory into which you downloaded Spark.

```
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master local \
  ./examples/jars/spark-examples_2.11-2.2.0.jar 10
```

What this will do is calculate the digits of pi to a certain level of estimation. What we've done here is specified that we want to run it on our local machine, specified which class and which jar we would like to run as well as any command line arguments to that particular class.

We can do this in Python with the following command line arguments.

```
./bin/spark-submit \
  --master local \
  ./examples/src/main/python/pi.py 10
```

By swapping out the path to the file and the cluster configurations, we can write and run production applications. Now Spark provides a lot more than just DataFrames that we can run as production applications. The rest of this chapter will walk through several different APIs that we can leverage to run all sorts of production applications.

Datasets: Type-Safe Structured APIs

The next topic we'll cover is a type-safe version of Spark's structured API for Java and Scala, called *Datasets*. This API is not available in Python and R, because those are dynamically typed languages, but it is a powerful tool for writing large applications in Scala and Java.

Recall that DataFrames, which we saw earlier, are a distributed collection of objects of type Row, which can hold various types of tabular data. The Dataset API allows users to assign a Java class to the records inside a DataFrame, and manipulate it as a collection of typed objects, similar to a Java `ArrayList` 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 where multiple software engineers must interact through well-defined interfaces.

The Dataset class is parametrized with the type of object contained inside: `Dataset<T>` in Java and `Dataset[T]` in Scala. As of Spark 2.0, the types `T` supported are all 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 inside your Dataset.

The awesome thing about Datasets is that we can use them only when we need or want to. For instance, in the follow example I'll define my own object and manipulate it via arbitrary map and filter functions. Once we've performed our manipulations, Spark can automatically turn it back into a DataFrame and we can manipulate it further 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. We cover this material extensively in the next part of this book, but here is a small example showing how we can use both type-safe functions and DataFrame-like SQL expressions to quickly write business logic.

```
%scala

// A Scala case class (similar to a struct) that will automatically
// be mapped into a structured data table in Spark
case class Flight(DEST_COUNTRY_NAME: String, ORIGIN_COUNTRY_NAME: String, count: BigInt)

val flightsDF = spark.read.parquet("/mnt/defg/flight-data/parquet/2010-summary.parquet/")
val flights = flightsDF.as[Flight]
```

One final advantage is that when you call `collect` or `take` on a Dataset, we're going to collect to objects of the proper type in your Dataset, not DataFrame Rows. This makes it easy to get type safety and safely perform manipulation in a distributed and a local manner without code changes.

```
%scala

flights
  .filter(flight_row => flight_row.ORIGIN_COUNTRY_NAME != "Canada")
  .take(5)
```

Structured Streaming

Structured Streaming is a high-level API for stream processing that became production-ready in Spark 2.2. Structured Streaming allows you to 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 get value out of streaming systems with virtually no code changes. It also makes it easy to reason about because you can write your batch job as a way to prototype it and then you can convert it to 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 where 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. Now 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.

```
InvoiceNo,StockCode,Description,Quantity,InvoiceDate,UnitPrice,CustomerID,Country
536365,85123A,WHITE HANGING HEART T-LIGHT HOLDER,6,2010-12-01 08:26:00,2.55,17850.0,United Kingdom
536365,71053,WHITE METAL LANTERN,6,2010-12-01 08:26:00,3.39,17850.0,United Kingdom
536365,84406B,CREAM CUPID HEARTS COAT HANGER,8,2010-12-01 08:26:00,2.75,17850.0,United Kingdom
```

Now in order 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 the Part V of this book.

```
%scala

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

staticDataFrame.createOrReplaceTempView("retail_data")
val staticSchema = staticDataFrame.schema
```

```
%python

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

staticDataFrame.createOrReplaceTempView("retail_data")
staticSchema = staticDataFrame.schema
```

Now since 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 largest sale hours where 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.

```
%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("InvoiceDate"), "1 day"))
    .sum("total_cost")
    .show(5)
```

```
%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("total_cost")\
    .show(5)
```

It's worth mentioning that we 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.

```
+----------+--------------------+------------------+
|CustomerId|              window|   sum(total_cost)|
+----------+--------------------+------------------+
|   17450.0|[2011-09-20 00:00...|          71601.44|
|      null|[2011-11-14 00:00...|          55316.08|
|      null|[2011-11-07 00:00...|          42939.17|
|      null|[2011-03-29 00:00...| 33521.39999999998|
|      null|[2011-12-08 00:00...|31975.590000000007|
+----------+--------------------+------------------+
```

The null values represent the fact that we don't have a customerId 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. Now we've seen how that works, let's take a look at the streaming code! You'll notice that very little actually changes about our code. The biggest change is that we used `readStream` instead of `read`, additionally you'll notice `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 be omitted.

Now since 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 simple specifies the number of partitions that should be created after a shuffle, by default the value is two hundred but since there aren't many executors on this machine it's worth reducing this to five. We did this same operation in the previous chapter, so if you don't remember why this is important feel free to flip back to the previous chapter to review.

```
val streamingDataFrame = spark.readStream
    .schema(staticSchema)
    .option("maxFilesPerTrigger", 1)
    .format("csv")
    .option("header", "true")
    .load("d/mnt/defg/retail-data/by-day/*.csv")
```

```
%python

streamingDataFrame = spark.readStream\
    .schema(staticSchema)\
    .option("maxFilesPerTrigger", 1)\
    .format("csv")\
    .option("header", "true")\
    .load("/mnt/defg/retail-data/by-day/*.csv")
```

Now we can see the 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.

```
%scala

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

```
%python

purchaseByCustomerPerHour = streamingDataFrame\
    .selectExpr(
    "CustomerId",
    "(UnitPrice * Quantity) as total_cost" ,
    "InvoiceDate" )\
    .groupBy(
    col("CustomerId"), window(col("InvoiceDate"), "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.

NOTE

Before kicking off the stream, we will set a small optimization that will allow this to run better on a single machine. This simply limits the number of output partitions after a shuffle, a concept we discussed in the last chapter. We discuss this in Part VI of the book.

```
spark.conf.set("spark.sql.shuffle.partitions", "5")
```

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 out 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 aggregation above.

```
%scala

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

```
%python

purchaseByCustomerPerHour.writeStream\
    .format("memory")\
    .queryName("customer_purchases")\
    .outputMode("complete")\
    .start()
```

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

```
%scala

spark.sql("""
    SELECT *
    FROM customer_purchases
    ORDER BY `sum(total_cost)` DESC
    """)
    .show(5)
```

```
%python

spark.sql("""
    SELECT *
    FROM customer_purchases
    ORDER BY `sum(total_cost)` DESC
    """)\
    .show(5)
```

You'll notice that as we read in more data - the composition of our table changes! With each file the results may or may not be changing based on the data. Naturally since 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 just simply write the results out to the console.

```
purchaseByCustomerPerHour.writeStream
  .format("console")
  .queryName("customer_purchases_2")
  .outputMode("complete")
  .start()
```

Neither of these streaming methods should be used 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 the data Spark processes the data. This was one of the shortcoming of Spark Streaming that Structured Streaming as resolved. We cover Structured Streaming in depth in Part V of this book.

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, clustering to deep learning. To demonstrate this functionality, we will perform some basic clustering on our data using a common algorithm called K-Means.

BOX What is K-Means? K-means is a clustering algorithm where "K" centers are randomly assigned within the data. The points closest to that point are then "assigned" to a particular cluster. Then a new center for this cluster is computed (called a 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 (where our centroid and clusters stop changing.

Spark includes a number of preprocessing methods out of the box. To demonstrate these methods, we will start 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.

```
staticDataFrame.printSchema()
```

```
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)
```

Machine learning algorithms in MLlib require data to be 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 will use several DataFrame transformations to manipulate our date data.

```
%scala
import org.apache.spark.sql.functions.date_format
val preppedDataFrame = staticDataFrame
    .na.fill(0)
    .withColumn("day_of_week", date_format($"InvoiceDate", "EEEE"))
    .coalesce(5)
```

```
%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)
```

Now we are also going to need to split our data into training and test sets. In this instance we are going to do this manually by the data that a certain purchase occurred however we could also leverage MLlib's transformation APIs to create a training and test set via train validation splits or cross validation. These topics are covered extensively in Part VI of this book.

```
%scala

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

```
%python

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

Now that we prepared our data, let's split it into a training and test set. Since this is a time-series set of data, we will split by an arbitrary date in the dataset. While this may 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()
```

Now these transformations are DataFrame transformations, covered extensively in part two of this book. Spark's MLlib also provides a number of transformations that allow us to automate some of our general transformations. One such transformer is a `StringIndexer`.

```
%scala

import org.apache.spark.ml.feature.StringIndexer

val indexer = new StringIndexer()
    .setInputCol("day_of_week")
    .setOutputCol("day_of_week_index")
```

```
%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 may 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. Therefore we need to use a `OneHotEncoder` 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.

```
%scala

import org.apache.spark.ml.feature.OneHotEncoder

val encoder = new OneHotEncoder()
    .setInputCol("day_of_week_index")
    .setOutputCol("day_of_week_encoded")
```

```
%python

from pyspark.ml.feature import OneHotEncoder

encoder = OneHotEncoder()\
    .setInputCol("day_of_week_index")\
    .setOutputCol("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 `Vector` type, which must be a set of numerical values.

```
%scala

import org.apache.spark.ml.feature.VectorAssembler


val vectorAssembler = new VectorAssembler()

  .setInputCols(Array("UnitPrice", "Quantity", "day_of_week_encoded"))

  .setOutputCol("features")
```

```
%python

from pyspark.ml.feature import VectorAssembler

vectorAssembler = VectorAssembler()\
  .setInputCols(["UnitPrice", "Quantity", "day_of_week_encoded"])\
  .setOutputCol("features")
```

We can see that we have 3 key features, the price, the quantity, and the day of week. Now we'll set this up into a pipeline so any future data we need to transform can go through the exact same process.

```
%scala

import org.apache.spark.ml.Pipeline

val transformationPipeline = new Pipeline()
    .setStages(Array(indexer, encoder, vectorAssembler))
```

```
%python

from pyspark.ml import Pipeline

transformationPipeline = Pipeline()\
    .setStages([indexer, encoder, vectorAssembler])
```

Now preparing for training is a two step process. We first need to fit our transformers to this dataset. We cover this in depth, but basically our `StringIndexer` needs to know how many unique values there are to be index. Once 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.

```
%scala

val fittedPipeline = transformationPipeline.fit(trainDataFrame)

%python

fittedPipeline = transformationPipeline.fit(trainDataFrame)
```

Once we fit the training data, we are now create to take that fitted pipeline and use it to transform all of our data in a consistent and repeatable way.

```
%scala

val transformedTraining = fittedPipeline.transform(trainDataFrame)
```

```
%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. At this point, we're going to perform some hyperparameter tuning on the model, since we do not want to repeat the exact same transformations over and over again, we'll leverage an optimization we discuss in Part IV of this book, caching. This will put a copy of this 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()
```

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

```
%scala

import org.apache.spark.ml.clustering.KMeans

val kmeans = new KMeans()
   .setK(20)
   .setSeed(1L)
```

```
%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, then we train it. There are always two types for every algorithm in MLlib's DataFrame API. They following the naming pattern of `Algorithm`, for the untrained version, and `AlgorithmModel` for the trained version. In our case, this is `KMeans` and then `KMeansModel`.

Predictors in MLlib's DataFrame API share roughly the same interface that we saw above 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. In our case we want to do things a bit more step by step, so we chose to not do this at this point.

```
%scala

val kmModel = kmeans.fit(transformedTraining)
```

```
%python

kmModel = kmeans.fit(transformedTraining)
```

We can see the resulting cost at this point. Which is quite high, that's likely because we didn't necessary scale our data or transform.

```
kmModel.computeCost(transformedTraining)
```

```
%scala

val transformedTest = fittedPipeline.transform(testDataFrame)
```

```
%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 of this book.

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 cover in the next chapter, 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 that DataFrames because they reveal physical execution characteristics (like partitions) to end users.

One thing you might use RDDs for is to parallelize raw data 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 can then convert that to a DataFrame to use it with other DataFrames.

```
%scala

spark.sparkContext.parallelize(Seq(1, 2, 3)).toDF()
```

```
%python

from pyspark.sql import Row

spark.sparkContext.parallelize([Row(1), Row(2), Row(3)]).toDF()
```

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 of this book. 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 where 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, we simply import it into our environment and run our 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.

```
%r

library(SparkR)

sparkDF <- read.df("/mnt/defg/flight-data/csv/2015-summary.csv",
         source = "csv", header="true", inferSchema = "true")

take(sparkDF, 5)
```

```
%r

collect(orderBy(sparkDF, "count"), 20)
```

R users can also leverage other R libraries like the pipe operator in magrittr in order 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.

```
%r

library(magrittr)

sparkDF %>%
    orderBy(desc(sparkDF$count)) %>%
    groupBy("ORIGIN_COUNTRY_NAME") %>%
    count() %>%
    limit(10) %>%
    collect()
```

We cover SparkR more in the Ecosystem Part of this book along with short discussion of PySpark specifics (PySpark is covered heavily through this book), and the new sparklyr package.

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. The list of packages is rather large at over 300 at the time of this writing and more are added frequently. The largest index of Spark Packages can be found at https://spark-packages.org/, where any user can publish to this package repository. There are also various other projects and packages that can be found through the web, for example on GitHub.

The Unified Analytics Platform

The datasets used in the book are also available for you to explore:
Spark: The Definitive Guide Datasets

Try Databricks for free
databricks.com/try-databricks

Contact us for a personalized demo
databricks.com/contact