Data Analysis with Python and PySpark

Jonathan Rioux

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Data Analysis with Python and PySpark
Version 7

For more information on this and other Manning titles go to
manning.com

welcome

Thank you for purchasing the MEAP for *Data Analysis with Python and PySpark.* It is a lot of fun (and work!) and I hope you'll enjoy reading it as much as I am enjoying writing the book.

My journey with PySpark is pretty typical: the company I used to work for migrated their data infrastructure to a data lake and realized along the way that their usual warehouse-type jobs didn't work so well anymore. I spent most of my first months there figuring out how to make PySpark work for my colleagues and myself, starting from zero. This book is very influenced by the questions I got from my colleagues and students (and sometimes myself). I've found that combining practical experience through real examples with a little bit of theory brings not only proficiency in using PySpark, but also how to build better data programs. This book walks the line between the two by explaining important theoretical concepts without being too laborious.

This book covers a wide range of subjects, since PySpark is itself a very versatile platform. I divided the book into three parts.

- Part 1: Walk teaches how PySpark works and how to get started and perform basic data manipulation.
- Part 2: Jog builds on the material contained in Part 1 and goes through more advanced subjects. It covers more exotic operations and data formats and explains more what goes on under the hood.
- Part 3: Run tackles the *cooler* stuff: building machine learning models at scale, squeezing more performance out of your cluster, and adding functionality to PySpark.

To have the best time possible with the book, you should be at least comfortable using Python. It isn't enough to have learned another language and transfer your knowledge into Python. I cover more niche corners of the language when appropriate, but you'll need to do some research on your own if you are new to Python.

Furthermore, this book covers how PySpark can interact with other data manipulation frameworks (such as Pandas), and those specific sections assume basic knowledge of Pandas.

Finally, for some subjects in Part 3, such as machine learning, having prior exposure will help you breeze through. It's hard to strike a balance between "not enough explanation" and "too much explanation"; I do my best to make the right choices.

Your feedback is key in making this book its best version possible. I welcome your comments and thoughts in the liveBook discussion forum.

Thank you again for your interest and in purchasing the MEAP!

—Jonathan Rioux

brief contents

1 Introduction

In this chapter, you will learn

- What is PySpark
- Why PySpark is a useful tool for analytics
- The versatility of the Spark platform and its limitations
- PySpark's way of processing data

According to pretty much every news outlet, data is everything, everywhere. It's the new oil, the new electricity, the new gold, plutonium, even bacon! We call it powerful, intangible, precious, dangerous. I prefer calling it *useful in capable hands*. After all, for a computer, any piece of data is a collection of zeroes and ones, and it is our responsibility, as users, to make sense of how it translates to something useful.

Just like oil, electricity, gold, plutonium and bacon (especially bacon!), our appetite for data is growing. So much, in fact, that computers aren't following. Data is growing in size and complexity, yet consumer hardware has been stalling a little. RAM is hovering for most laptops at around 8 to 16 GB, and SSD are getting prohibitively expensive past a few terabytes. Is the solution for the burgeoning data analyst to triple-mortgage his life to afford top of the line hardware to tackle Big Data problems?

Introducing Spark, and its companion PySpark, the unsung heroes of large-scale analytical workloads. They take a few pages of the supercomputer playbook— powerful, but manageable, compute units meshed in a network of machines— and bring it to the masses. Add on top a powerful set of data structures ready for any work you're willing to throw at them, and you have a tool that will *grow* (pun intended) with you.

This book is great introduction to data manipulation and analysis using PySpark. It tries to cover

just enough theory to get you comfortable, while giving enough opportunities to practice. Each chapter except this one contains a few exercices to anchor what you just learned. The exercises are all solved and explained in Appendix A.

1.1 What is PySpark?

What's in a name? Actually, quite a lot. Just by separating PySpark in two, one can already deduce that this will be related to Spark and Python. And it would be right!

At the core, PySpark can be summarized as being the Python API to Spark. While this is an accurate definition, it doesn't give much unless you know the meaning of Python and Spark. If we were in a video game, I certainly wouldn't win any prize for being the most useful NPC. Let's continue our quest to understand what is PySpark by first answering *What is Spark?*.

1.1.1 You saw it coming: What is Spark?

Spark, according to their authors, is a *unified analytics engine for large-scale data processing*. This is a very accurate definition, if a little dry.

Digging a little deeper, we can compare Spark to an *analytics factory*. The raw material— here, data— comes in, and data, insights, visualizations, models, you name it! comes out.

Just like a factory will often gain more capacity by increasing its footprint, Spark can process an increasingly vast amount of data by *scaling out* instead of *scaling up*. This means that, instead of buying thousand of dollars of RAM to accommodate your data set, you'll rely instead of multiple computers, splitting the job between them. In a world where two modest computers are less costly than one large one, it means that scaling out is less expensive than up, keeping more money in your pockets.

The problem with computers is that they crash or behave unpredictably once in a while. If instead of one, you have a hundred, the chance that at least one of them go down is now much higher.[1] Spark goes therefore through a lot of hoops to manage, scale, and babysit those poor little sometimes unstable computers so you can focus on what you want, which is to work with data.

This is, in fact, one of the weird thing about Spark: it's a good tool because of what you can do with it, but especially because of what you *don't have to do* with it. Spark provides a powerful API[2] that makes it look like you're working with a cohesive, non-distributed source of data, while working hard in the background to optimize your program to use all the power available. You therefore don't have to be an expert at the arcane art of distributed computing: you just need to be familiar with the language you'll use to build your program. This leads us to…

1.1.2 PySpark = Spark + Python

PySpark provides an entry point to Python in the computational model of Spark. Spark itself is coded in Scala, a language very powerful if a little hard to grasp. In order to meet users where they are, Spark also provides an API in Java, Python and R. The authors did a great job at providing a coherent interface between language while preserving the idiosyncrasies of the language where appropriate. Your PySpark program will therefore by quite easy to read by a Scala/Spark programmer, but also to a fellow Python programmer who hasn't jumped into the deep end (yet).

Python is a dynamic, general purpose language, available on many platforms and for a variety of tasks. Its versatility and expressiveness makes it an especially good fit for PySpark. The language is one of the most popular for a variety of domains, and currently is a major force in data analysis and science. The syntax is easy to learn and read, and the amount of library available means that you'll often find one (or more!) who's just the right fit for your problem.

1.1.3 Why PySpark?

There are no shortage of libraries and framework to work with data. Why should one spend their time learning PySpark specifically?

PySpark packs a lot of advantages for modern data workloads. It sits at the intersection of fast, expressive and versatile. Let's explore those three themes one by one.

PYSPARK IS FAST

If you search "Big Data" in a search engine, there is a very good chance that Hadoop will come within the first few results. There is a very good reason to this: Hadoop popularized the famous *MapReduce* framework that Google pioneered in 2004 and is now a staple in Data Lakes and Big Data Warehouses everywhere.

Spark was created a few years later, sitting on Hadoop's incredible legacy. With an aggressive query optimizer, a judicious usage of RAM and some other improvements we'll touch on in the next chapters, Spark can run up to 100x faster than plain Hadoop. Because of the integration between the two frameworks, you can easily switch your Hadoop workflow to Spark and gain the performance boost without changing your hardware.

PYSPARK IS EXPRESSIVE

Beyond the choice of the Python language, one of the most popular and easy-to-learn language, PySpark's API has been designed from the ground up to be easy to understand. Most programs read as a descriptive list of the transformations you need to apply to the data, which makes them easy to reason about. For those familiar with functional programming languages, PySpark code is conceptually closer to the "pipe" abstraction rather than pandas, the most popular in-memory DataFrame library.

You will obviously see many examples through this book. As I was writing those examples, I was pleased about how close to my initial (pen and paper) reasoning the code ended up looking. After understanding the fundamentals of the framework, I'm confident you'll be in the same situation.

PYSPARK IS VERSATILE

There are two components to this versatility. First, there is the *availability* of the framework. Second, there is the diversified *ecosystem* surrounding Spark.

PySpark is everywhere. All three major cloud providers have a managed Hadoop/Spark cluster as part of their offering, which means you have a fully provisioned cluster at a click of a few buttons. You can also easily install Spark on your own computer to nail down your program before scaling on a more powerful cluster. Appendix B covers how to get your own local Spark running, while Appendix C will walk through the current main cloud offerings.

PySpark is open-source. Unlike some other analytical software, you aren't tied to a single company. You can inspect the source code if you're curious, and even contribute if you have an idea for a new functionality or find a bug. It also gives a low barrier to adoption: download, learn, profit!

Finally, Spark's eco-system doesn't stop at PySpark. There is also an API for Scala, Java, R, as well as a state-of-the-art SQL layer. This makes it easy to write a polyglot program in Spark. A Java software engineer can tackle the ETL pipeline in Spark using Java, while a data scientist can build a model using PySpark.

WHERE PYSPARK FELL SHORT

It would be awesome if PySpark was The Answer to every data problem. Unfortunately, there are some caveats. None of them are a deal-breakers, but they are to be considered when you're selecting a framework for your next project.

PySpark isn't the right choice if you're dealing with small data sets. Managing a distributed cluster comes with some overhead, and if you're just using a single node, you're paying the price but aren't using the benefits. As an example, a PySpark shell will take a few seconds to launch:

this is often more than enough time to process data that fits within your RAM.

PySpark also has a disadvantage when it comes to the Java and Scala API. Since Spark is at the core a Scala program, Python code have to be translated to and from JVM[3] instructions. While more recent versions have been bridging that gap pretty well, pure Python translation, which happens mainly when you're defining your own User Defined Functions (UDF), will perform slower. We will cover UDF and some ways to mitigate the performance problem in Chapter 8.

Finally, while programming PySpark can feel easy and straightforward, managing a cluster can be a little arcane. Spark is a pretty complicated piece of software, and while the code base matured remarkably over the past few years, the days where scaling a 100-machine cluster and manage it as easily as a single node are far ahead. We will cover some of the developer-facing configuration and problems in the Chapter about performance, but for hairier problems, do what I do: befriend your dev ops.

1.1.4 Your very own factory: how PySpark works

In this section, I will explain how Spark processes a program. It can be a little odd to present the workings and underpinnings of a system that we claimed, a few paragraphs ago, hides that complexity. We still think that having a working knowledge of how Spark is set up, how it manages data and how it optimizes queries is very important. With this, you will be able to reason with the system, improve your code and figure out quicker when it doesn't perform the way you want.

If we're keeping the factory analogy, we can imagine that the cluster of computer where Spark is sitting on is the building.

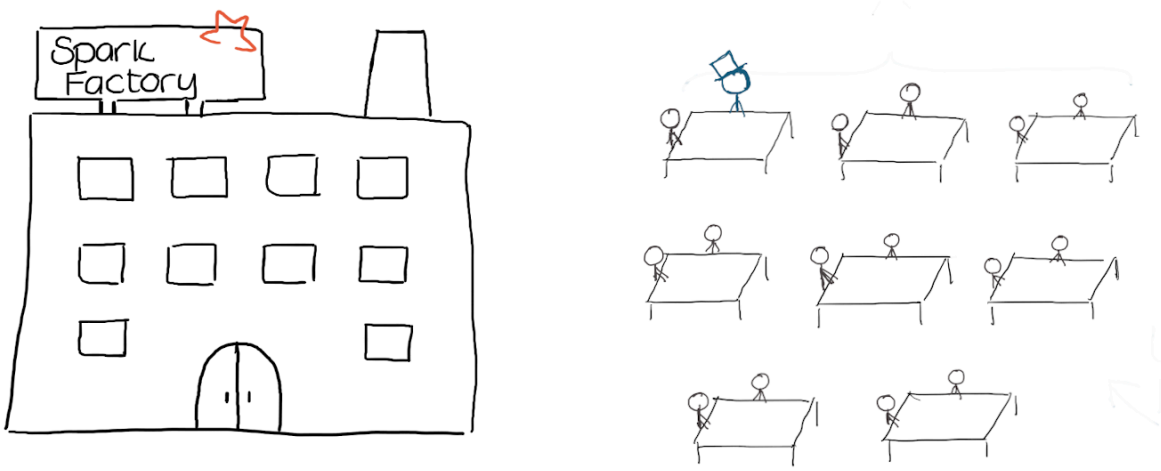


Figure 1.1 A totally relatable data factory, outside and in.

If we look at , we can see two different way to interpret a data factory. On the left, we see how it looks like from the outside: a cohesive unit where projects come in and results comes out. This is what it will appear to you most of the time. Under the hood, it looks more like on the right: you have some workbenches where some workers are assigned to. The workbenches are like the computers in our Spark cluster: there is a fixed amount of them, and adding or removing some is easy but needs to be planned in advance. The workers are called *executors* in Spark's literature: they are the one performing the actual work on the machines.

One of the little workers looks spiffier than the other. That top hat definitely makes him stand out of the crowd. In our data factory, he's the manager of the work floor. In Spark terms, we call this the *master*. In the spirit of the open work-space, he shares one of the workbenches with his fellow employees. The role of the master is crucial to the efficient execution of your program, so is dedicated to this.

1.1.5 Some physical planning with the cluster manager

Upon reception of the task, which is called a *driver program* in the Spark world, the factory starts running. This doesn't mean that we get straight to processing! Before that, the cluster need to *plan the capacity* it will allocate for your program. The entity, or program, taking care of this is aptly called the *cluster manager*. In our factory, this cluster manager will look at the workbenches with available space and secure as many as necessary, then start hiring workers to fill the capacity. In Spark, it will look at the machines with available computing resources and secure what's necessary, before launching the required number of executors across them.

> **NOTE** Spark provides its own cluster manager, but can also play well with other ones when working in conjunction with Hadoop or another Big Data platform. We will definitely discuss the intricacies of managing the cluster manager (pun intended) in the chapter about performance, but in the meantime, if you read about YARN or Mesos in the wild, know that they are two of the most popular nowadays.

Any directions about capacity (machines and executors) are encoded in a `SparkContext` object which represents the connection to our Spark cluster. Since our instructions didn't mention any specific capacity, the cluster manager will allocate the default capacity prescribed by our Spark installation.

We're off to a great start! We have a task to accomplish, and the capacity to accomplish it. What's next? Let's get working!

1.1.6 A factory made efficient through a lazy manager

Just like in a large-scale factory, you don't go to each employee and give them a list of tasks. No, here, you'll *provide your list of steps to the manager* and let them deal with it. In Spark, the manager/*master* takes your instructions (carefully written in Python code), translate them in Spark steps and then process them across the worker. The master will also manage which *worker* (more on them in a bits) has which slice of the data, and make sure that you don't lose some bits in the process.

Your manager/master has all the qualities a good manager has: smart, cautious and lazy. Wait, what? You read me right. *Laziness* in a programming context— and one could argue in the real world too— can actually be very good thing. Every instruction you're providing in Spark can be classified in two categories: transformations and actions. *Actions* are what many programming languages would consider IO. Actions includes, but are not limited to:

- Printing information on the screen
- Writing data to a hard drive or cloud bucket

In Spark, we'll see those instructions most often via the `show` and `write` methods, as well as other calling those two in their body.

Transformations are pretty much everything else. Some examples of transformation are:

- Adding a column to a table
- Performing an aggregation according to certain keys
- Computing summary statistics on a data set
- Training a Machine Learning model on some data

Why the distinction, you might ask? When thinking about computation over data, you, as the developer, are only concerned about the computation leading to an action. You'll always interact with the results of an action, because this is something you can see. Spark, with his lazy computation model, will take this to the extreme and will avoid performing data work until an action triggers the computation chain. Before that, the master will store (or *cache*) your instructions. This way of dealing with computation has many benefits when dealing with large scale data.

First, storing instructions in memory takes much less space than storing intermediate data results. If you are performing many operations on a data set and are materializing the data each step of the way, you'll blow your storage much faster although you don't need the intermediate results. We can all argue that less waste is better.

Second, by having the full list of tasks to be performed available, the master can optimize the work between executors much more efficiently. It can use information available at run-time, such as the node where specific parts of the data are located. It can also re-order and eliminate useless

transformations if necessary.

Finally, during interactive development, you don't have to submit a huge block of commands and wait for the computation to happen. Instead, you can iteratively build your chain of transformation, one at the time, and when you're ready to launch the computation (like during your coffee break), you can add an action and let Spark work its magic.

Lazy computation is a fundamental aspect of Spark's operating model and part of the reason it's so fast. Most programming languages, including Python, R and Java, are eagerly evaluated. This means that they process instructions as soon as they receive them. If you have never worked with a lazy language before, it can look a little foreign and intimidating. If this is the case, don't worry: we'll weave practical explanations and implications of that laziness during the code examples when relevant. You'll be a lazy pro in no time!

NOTE Reading data, although clearly being I/O, is considered a transformation by Spark. This is due to the fact that reading data doesn't perform any visible work to the user. You therefore won't read data until you need to display or write it somewhere.

What's a manager without competent employees? Once the task, with its action, has been received, the master starts allocating data to what Spark calls *executors*. Executors are processes that run computations and store data for the application. Those executors sit on what's called a *worker node*, which is the actual computer. In our factory analogy, an executor would be an employee performing the work, while the worker node would be a workbench where many employees/executors can work. If we recall , our master wears a top hat and sits with his employees/workers at one of the workbenches.

That concludes our factory tour. Let's summarize our typical PySpark program.

We first encode our instructions in Python code, forming a driver program.

When submitting our program (or launching a PySpark shell), the cluster manager allocates resources for us to use. Those will stay constant for the duration of the program.

The master ingests your code and translate it into Spark instructions. Those instructions are either transformations or actions.

Once the master reaches an action, it optimizes the whole computation chain and splits the work between executors. Executors are processes performing the actual data work and they reside on machines labelled worked nodes.

That's it! As we can see, the overall process is quite simple, but it's obvious that Spark hides a lot of the complexity arising from efficient distributed processing. For a developer, this means

shorter and clearer code, and a faster development cycle.

1.2 What will you learn in this book?

This book will use PySpark to solve a variety of tasks a data analyst, engineer or scientist will encounter during his day to day life. We will therefore

- read and write data from (and to) a variety of sources and formats;
- deal with messy data with PySpark's data manipulation functionality;
- discover new data sets and perform exploratory data analysis;
- build data pipelines that transform, summarize and get insights from data in an automated fashion;
- test, profile and improve your code;
- troubleshoot common PySpark errors, how to recover from them and avoid them in the first place.

After covering those fundamentals, we'll also tackle different tasks that aren't as frequent, but are interesting and an excellent way to showcase the power and versatility of PySpark.

- We'll perform Network Analysis using PySpark's own graph representation
- We'll build Machine Learning models, from simple throwaway experiments to Deep Learning goodness
- We'll extend PySpark's functionality using user defined functions, and learn how to work with other languages

We are trying to cater to many potential readers, but are focusing on people with little to no exposure to Spark and/or PySpark. More seasoned practitioners might find useful analogies for when they need to explain difficult concepts and maybe learn a thing or two!

The book focuses on Spark version 2.4, which is currently the most recent available. Users on older Spark versions will be able to go through most of the code in the book, but we definitely recommend using at least Spark 2.0+.

We're assuming some basic Python knowledge: some useful concepts are outlined in Appendix D. If you feel for a more in-depth introduction to Python, I recommend *The Quick Python Book*, by Naomi Ceder (Manning, 2018).

1.3 What do I need to get started?

In order to get started, the only thing absolutely necessary is a working installation of Spark. It can be either on your computer (Appendix B) or using a cloud provider (Appendix C). Most examples in the book are doable using a local installation of Spark, but some will require more horsepower and will be identified as such.

A code editor will also be very useful for writing, reading and editing scripts as you go through

the examples and craft your own programs. A Python-aware editor, such as PyCharm, is a nice-to-have but is in no way necessary. Just make sure it saves your code without any formatting: don't use Microsoft Word to write your programs!

The book's code examples are available on GitHub, so Git will be a useful piece of software to have. If you don't know git, or don't have it handy, GitHub provides a way to download all the book's code in a Zip file. Make sure you check regularly for updates!

Finally, I recommend that you have an analog way of drafting your code and schema. I am a compulsive note-taker and doodler, and even if my drawing are very basic and crude, I find that working through a new piece of software via drawings helps in clarifying my thoughts. This means less code re-writing, and a happier programmer! Nothing spiffy, some scrap paper and a pencil will do wonders.

1.4 Summary

- PySpark is the Python API for Spark, a distributed framework for large-scale data analysis. It provides the expressiveness and dynamism of the Python programming language to Spark.
- PySpark provides a full-stack analytics workbench. It has an API for data manipulation, graph analysis, streaming data as well as machine learning.
- Spark is fast: it owes its speed to a judicious usage of the RAM available and an aggressive and lazy query optimizer.
- Spark provides bindings for Python, Scala, Java, and R. You can also use SQL for data manipulation.
- Spark uses a *master* which processes the instructions and orchestrates the work. The *executors* receive the instructions from the master and perform the work.
- All instructions in PySpark are either transformations or actions. Spark being lazy, only actions will trigger the computation of a chain of instructions.

2 Your first data program in PySpark

This chapter covers:

- Launching and using the `pyspark` shell for interactive development
- Reading and ingesting data into a data frame
- Exploring data using the `DataFrame` structure
- Selecting columns using the `select()` method
- Filtering columns using the `where()` method
- Applying simple functions to your columns to modify the data they contain
- Reshaping singly-nested data into distinct records using `explode()`

Data-driven applications, no matter how complex, all boils down to what I like to call three *meta-steps*, which are easy to distinguish in a program.

1. We start by *ingesting* or reading the data we wish to work with.
2. We *transform* the data, either via a few simple instructions or a very complex machine learning model
3. We then *export* the resulting data, either into a file to be fed into an app or by summarizing our findings into a visualization.

The next two chapters will introduce a basic workflow with PySpark via the creation of a simple ETL (*Extract, Transform and Load*, which is a more business-speak way of saying *Ingest, Transform and Export*). We will spend most of our time at the `pyspark` shell, interactively building our program one step at a time. Just like normal Python development, using the shell or REPL (I'll use the terms interchangeably) provides rapid feedback and quick progression. Once we are comfortable with the results, we will wrap our program so we can submit it in batch mode.

Data manipulation is the most basic and important aspect of any data-driven program and PySpark puts a lot of focus on this. It serves as the foundation of any reporting, any machine

learning or data science exercise we wish to perform. This section will give you the tools to not only use PySpark to manipulate data at scale, but also how to *think* in terms of data transformation. We obviously can't cover every function provided in PySpark, but I'll provide a good explanation of the ones we use. I'll also introduce how to use the shell as a friendly reminder for those cases when you forget how something works.

Since this is your first end-to-end program in PySpark, we'll get our feet wet with a simple problem to solve: counting the most popular word being used in the English language. Now, since collecting all the material ever produced in the English language would be a massive undertaking, we'll start with a very small sample: *Pride and Prejudice* by Jane Austen. We'll make our program work with this small sample and then scale it to ingest a larger corpus of text.

Since this is our first program, and I need to introduce many new concepts, this Chapter will focus on the data manipulation part of the program. Chapter 3 will cover the final computation as well as wrapping our program and then scaling it.

> **TIP** The book repository, containing the code and data used for the examples and exercises, is available at https://github.com/jonesberg/PySparkInAction .

2.1 Setting up the pyspark shell

PySpark provides a REPL (*Read, eval, print loop*) for interactive development. Python and other programming language families, such as Lisp, also provides one, so there is a good chance that you already worked with one in the past. It speeds up your development process by giving instantaneous feedback the moment you submit an instruction, instead of forcing you to compile your program and submit it as one big monolithic block. I'll even say that using a REPL is even more useful in PySpark, since every operation can take a fair amount of time. Having a program crash mid-way is always frustrating, but it's even worse when you've been running a data intensive job for a few hours.

For this chapter (and the rest of the book), I assume that you have access to a working installation of Spark, either locally or in the cloud. If you want to perform the installation yourself, Appendix B contains step-by-step instructions for Linux, OsX and Windows. If you can't install it on your computer, or prefer not to, Appendix C provides a few cloud-powered options as well as additional instructions to upload your data and make it visible to Spark.

Once everything is set up, you can launch the `pyspark` shell by inputting `pyspark` into your terminal. You should see an ASCII-art version of the Spark logo, as well as some useful information. shows what happens on my local machine.

Listing 2.1 Launching `pyspark` on a local machine

```
$ pyspark

Python 3.7.3 (default, Mar 27 2019, 16:54:48)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.4.0 -- An enhanced Interactive Python. Type '?' for help.
19/09/07 12:16:47 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using
  ❶
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).   ❷
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 2.4.3   ❸
      /_/

Using Python version 3.7.3 (default, Mar 27 2019 16:54:48)   ❹
SparkSession available as 'spark'.   ❺

In [1]:   ❻
```

❶ When using PySpark locally, you most often won't have a full Hadoop cluster pre-configured. For learning purposes, this is perfectly fine.

❷ Spark is indicating the level of details it'll provide to you. We will see how to configure this in .

❸ We are using Spark version 2.4.3

❹ PySpark is using the Python available on your path.

❺ The `pyspark` shell provides an entry point for you through the variable `spark`. More on this in .

❻ The REPL is now ready for your input!

> **NOTE** I highly recommend you using IPython when using PySpark in interactive mode. IPython is a better front-end to the Python shell containing many useful functionalities, such as friendlier copy-and-paste and syntax highlighting. The installation instructions in Appendix B includes configuring PySpark to use the IPython shell.

While all the information provided in is useful, two elements are worth expanding on: the `SparkSession` entry point and the log level.

2.1.1 The `SparkSession` entry-point

In we saw that, upon launching the PySpark shell creates a `spark` variable that refers to `SparkSession` entry point. I will discuss about this entry point in this section as it provides the functionality for us to read data into PySpark[4].

In Chapter 1, we spoke briefly about the Spark entry point called `SparkContext`. `SparkSession` is a super-set of that. It wraps the `SparkContext` and provides functionality for interacting with data in a distributed fashion.Just to prove our point, see how easy it is to get to the `SparkContext` from our `SparkSession` object: just call the `sparkContext` attribute from `spark`.

```
$ spark.sparkContext
# <SparkContext master=local[*] appName=PySparkShell>
```

The `SparkSession` object is a recent addition to the PySpark API, making its way in version 2.0. This is due to the API evolving in a way that makes more room for the faster, more versatile data frame as the main data structure over the lower level RDD. Before that time, you had to use another object (called the `SQLContext`) in order to use the data frame. It's much easier to have everything under a single umbrella.

This book will focus mostly on the data frame as our main data structure. I'll discuss about the RDD in Chapter 8, when we discuss about lower-level PySpark programming and how to embed our own Python functions in our programs.

SIDEBAR **Reading older PySpark code**

While this book shows modern PySpark programming, we are not living in a vacuum. If you go on the web, you might face older PySpark code that uses the former `SparkContext`/ `sqlContext` combo. You'll also see the `sc` variable mapped to the `SparkContext` entry-point. With that we know about `SparkSession` and `SparkContext`, we can reason about old PySpark code by using the following variable assignments.

```
sc = spark.sparkContext
sqlContext = spark
```

You'll see traces of `SQLContext` in the API documentation for backwards compatibility. I recommend avoiding using this as the new `SparkSession` approach is cleaner, simpler and more future-proof.

2.1.2 Configuring how chatty spark is: the log level

Monitoring your PySpark jobs is an important part of developing a robust program. PySpark provides many levels of logging, from nothing at all to a full description of everything happening on the cluster. By default, the `pyspark` shell defaults on `WARN`, that can be a little chatty when we're learning. Fortunately, we can change the settings for your session by using the code in .

Listing 2.2 Deciding on how chatty you want PySpark to be.

```
spark.sparkContext.setLogLevel(KEYWORD)
```

lists the available keywords you can pass to `setLogLevel`. Each subsequent keyword contains all the previous ones, with the obvious exception of `OFF` that doesn't show anything.

Table 2.1 log level keywords

Keyword	Signification
`OFF`	No logging at all (not recommended).
`FATAL`	Only fatal errors
`ERROR`	My personal favorite, will show `FATAL` as well as other useful (but recoverable) errors.
`WARN`	Add warnings (and there is quite a lot of them).
`INFO`	Will give you runtime information
`DEBUG`	Will provide debug information on your jobs.
`TRACE`	Will trace your jobs (more verbose debug logs). Can be quite pedagogic, but very annoying.
`ALL`	Everything that PySpark can spit, it will spit.

NOTE When using the `pyspark` shell, anything chattier than `WARN` might appear when you're typing a command, which makes it quite hard to input commands into the shell. You're welcome to play with the log levels as you please, but we won't show any output unless it's valuable for the task at hand.

Setting the log level to `ALL` is a *very* good way to annoy oblivious co-workers if they don't lock their computers. You haven't heard it from me.

You now have the REPL fired-up and ready for your input.

2.2 Mapping our program

In the Chapter introduction, we introduced our problem statement: *what are the most popular words in the English language?* Before even hammering code in the REPL, we can start by mapping the major steps our program will need to perform.

1. *Read*: Read the input data (we're assuming a plain text file)
2. *Token*: Tokenize each word
3. *Clean*: Remove any punctuation and/or tokens that aren't words.
4. *Count*: Count the frequency of each word present in the text
5. *Answer:* Return the top 10 (or 20, 50, 100)

Visually, a simplified flow of our program would look like

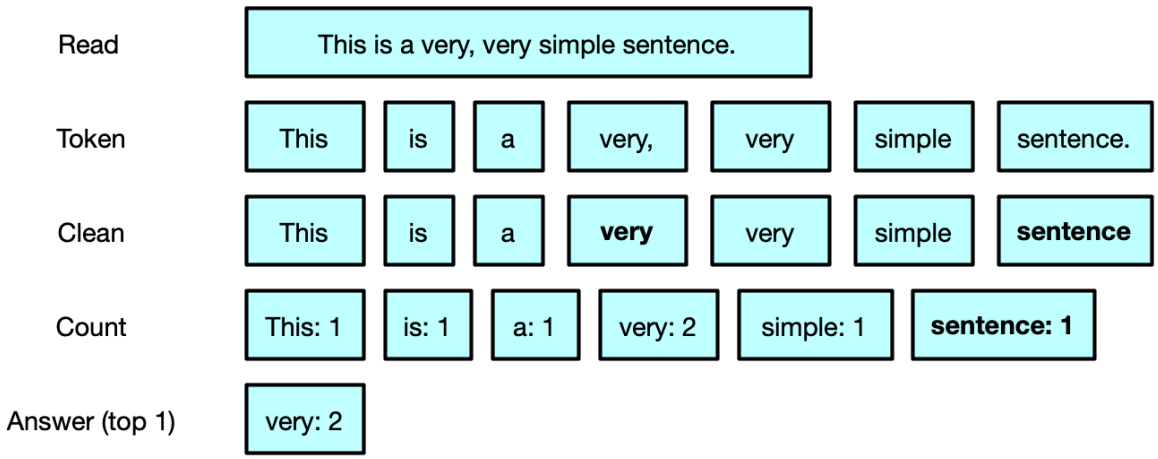


Figure 2.1 A simplified flow of our program, illustrating the 5 steps.

Our goal is quite lofty: the English language produced through history an unfathomable amount of written material. Since we are learning, we'll start with a relatively small source, get our program working, and then scale it to accommodate a larger body of text. For this, I chose to use Jane Austen's *Pride and Prejudice*, since it's already in plain text and freely available.

SIDEBAR **Data analysis and Pareto's principle**

Pareto's principle, also known commonly as the 80/20 rules, is often summarized as *20% of the efforts will yield 80% of the results.* In data analysis, we can consider that 20% to be analysis, visualization, machine learning models, anything that provides tangible value to the recipient.

The remainder is what I call *invisible work*: ingesting the data, cleaning it, figuring its meaning and shaping it into a usable form. If you look at your simple steps, Step 1 to 3 can be considered invisible work: we're ingesting data and getting it ready for the counting process. Step 4 and 5 are really the visible ones that are answering our question (one could argue that only Step 5 is performing visible work, but let's not split hairs here). Steps 1 to 3 are there because the data requires processing to be usable for our problem. They aren't core to our problem, but we can't do without them.

When building your own project, this will be the part that will be the most time consuming and you might be tempted (or pressured!) to skimp on it. Always keep in mind that the data you ingest and process is the raw material of your programs, and that feeding it garbage will yield, well, garbage.

2.3 Reading and ingesting data into a data frame

The first step of our program is to ingest the data in a structure we can perform work in. PySpark provides two main structures for performing data manipulation:

1. The Resilient Distributed Dataset (or RDD)

2. The data frame

The RDD can be seen like a distributed collection of objects. I personally visualize this as a bag that you give orders to. You pass orders to the RDD through regular Python functions over the items in the bag.

The data frame is a stricter version of the RDD: conceptually, you can think of it like a table, where each cell can contain one value. The data frame makes heavy usage of the concept of *columns* where you perform operation on columns instead of on records, like in the RDD. provides a visual summary of the two structures.

If you've used SQL in the past, you'll find that the data frame implementation takes a lot of inspiration from SQL. The module name for data organization and manipulation is even named `pyspark.sql`! Furthermore, Chapter 7 will teach you how to mix PySpark and SQL code within the same program.

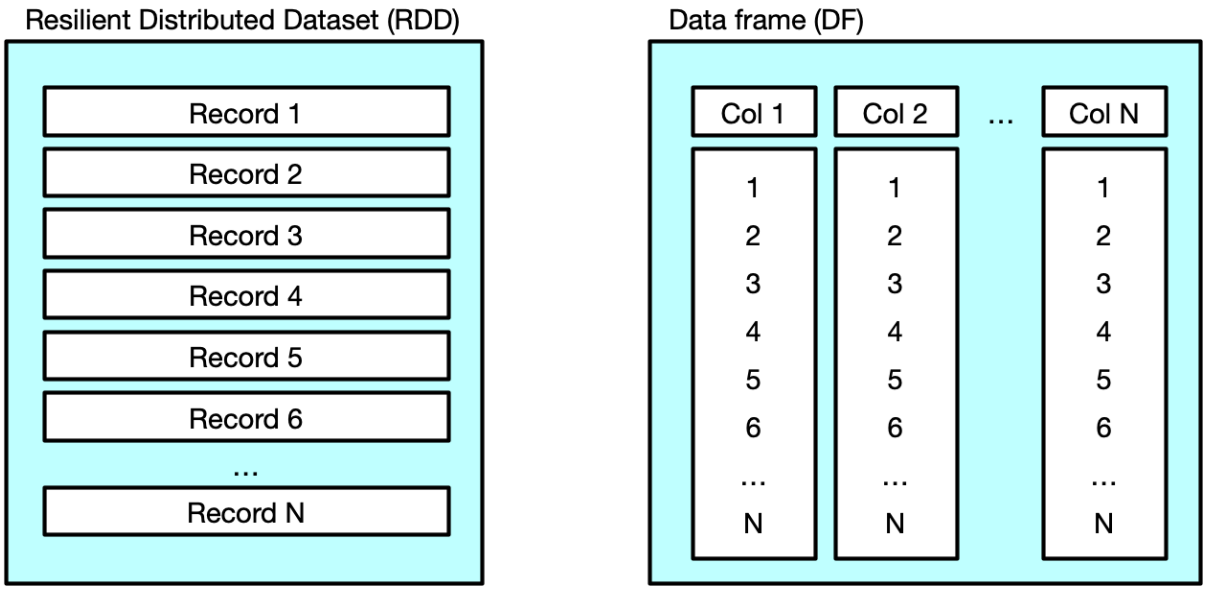


Figure 2.2 A RDD vs a data frame. In the RDD, each record is processed independently. With the data frame, we work with its columns, performing functions on them.

SIDEBAR **Some language convention**

Since this book will talk about data frames more than anything else, I prefer using the non-capitalized nomenclature, *i.e.* "data frame". I find this to be more readable than using capital letters or even DataFrame without a space.

When referring to the PySpark object directly, I'll use `DataFrame` but with a fixed-width font. This will help differentiate between data frame the concept and `DataFrame` the object.

This book will focus on the data frame implementation as it is more modern and performs faster

for all but the most esoteric tasks. Chapter 8 will discuss about trade-offs between the RDD and the data frame. Don't worry: once you're learned the data frame, it'll be a breeze the learn the RDD.

Reading data into a data frame is done through the `DataFrameReader` object, which we can access through `spark.read`. The code in displays the object, as well as the methods it exposes. We recognize a few file formats: `csv` stands for comma separated values (which we'll use as early as Chapter 4), `json` for JavaScript Object Notation (a popular data exchange format) and `text` is plain text.

Listing 2.3 The `DataFrameReader` object

```
In [3]: spark.read
Out[3]: <pyspark.sql.readwriter.DataFrameReader at 0x115be1b00>

In [4]: dir(spark.read)
Out[4]: [<some content removed>, _spark', 'csv', 'format', 'jdbc', 'json',
'load', 'option', 'options', 'orc', 'parquet', 'schema', 'table', 'text']
```

SIDEBAR **PySpark reads your data**

PySpark provides many readers to accommodate the different ways you can process data. Under the hood, `spark.read.csv()` will map to `spark.read.format('csv').load()` and you may encounter this form in the wild. I usually prefer using the direct `csv` method as it provides a handy reminder of the different parameters the reader can take.

`orc` and `parquet` are also data format especially well suited for big data processing. ORC (which stands for *Optimized Row Columnar*) and Parquet are competing data format which serves pretty much the same purpose. Both are open-sourced and now part of the Apache project, just like Spark.

PySpark defaults to using parquet when reading and writing files, and we'll use this format to store our results through the book. I'll provide a longer discussion about the usage, advantages and trade-offs of using Parquet or ORC as a data format in Chapter 6.

Let's read our data file. I am assuming that you launched PySpark at the root of this book's repository. Depending on your case, you might need to change the path where the file is located.

Listing 2.4 "Reading" our Jane Austen novel in record time

```
book = spark.read.text("./data/Ch02/1342-0.txt")

book
# DataFrame[value: string]
```

We get a data frame, as expected! If you input your data frame, conveniently named `book`, into the shell, you see that PySpark doesn't actually output any data to the screen. Instead, it prints

the schema, which is the name of the columns and their type. In PySpark's world, each column has a type: it represents how the value is represented by Spark's engine. By having the type attached to each column, you can know instantly what operations you can do on a the data. With this information, you won't inadvertently try to add an integer to a string: PySpark won't let you add 1 to "blue". Here, we have one column, named `value`, composed of a `string`. A quick graphical representation of our data frame would look like . Besides being a helpful reminder of the content of the data frame, types are integral to how Spark processes data quickly and accurately. We will explore the subject extensively in Chapter 5.

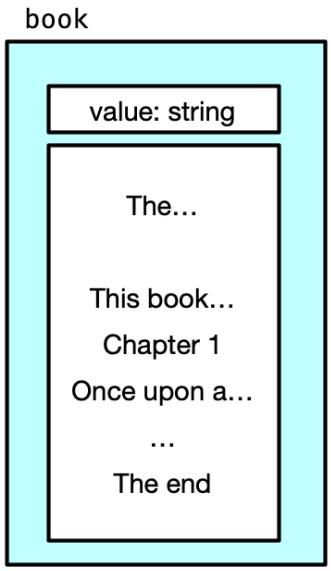


Figure 2.3 A high-level schema of a our book data frame, containing a value string column. We can see the name of the column, its type, and a small snippet of the data.

If you want to see the schema in a more readable way, you can use the handy method `printSchema()`, illustrated in . This will print a tree-like version of the data frame's schema. It is probably the method I use the most when developing interactively!

Listing 2.5 Printing the schema of our data frame

```
book.printSchema()

# root
#  |-- value: string (nullable = true)
```

Same information, displayed in a friendlier way.

SIDEBAR **Speeding up your learning by using the shell**

This doesn't just apply to PySpark, but using the functionality of the shell can often save a lot of searching into the documentation. I am a big fan of using `dir()` on an object when I don't remember the exact method I want to apply, like I did in .

PySpark's source code is very well documented, if you're unsure about the proper usage of a function, class or method, you can print the *`doc`* attribute or, for those using IPython, use a trailing question mark (or two, if you want more details).

Listing 2.6 Using PySpark's documentation directly in the REPL

```
In [*]: print(spark.read.__doc__)

    Interface used to load a :class:`DataFrame` from external storage systems
    (e.g. file systems, key-value stores, etc). Use :func:`spark.read`
    to access this.

    .. versionadded:: 1.4


In [*]: spark.read?
Type:        property
String form: <property object at 0x1159a0958>
Docstring:
Returns a :class:`DataFrameReader` that can be used to read data
in as a :class:`DataFrame`.

:return: :class:`DataFrameReader`

.. versionadded:: 2.0
```

2.4 Exploring data in the `DataFrame` structure

One of the key advantages of using the REPL for interactive development is that you can peek at your work as you're performing it. Now that our data is loaded into a data frame, we can start looking at how PySpark structured our text.

In , we saw that the default behaviour of imputing a data frame in the shell is to provide the schema or column information of the object. While very useful, sometimes we want to take a peek of the data.

Enter the `show()` method.

2.4.1 Peeking under the hood: the `show()` method

The most fundamental operation of any data processing library or program is displaying the data it contains. In the case of PySpark, it becomes even more important, since we'll definitely be working with data that goes beyond a screenful. Still, sometimes, you just want to see the data, raw, without any complications.

The `show()` method displays a sample of the data back to you. Nothing more, nothing less. With `printSchema()`, it will become one of your best friend to perform data exploration and validation. By default, it will show 20 rows and truncate long rows. The code in shows the default behaviour of the method, applied to our `book` data frame. For text data, the length limitation is limiting (pun intended). Fortunately, `show()` provides some options to display just what you need.

Listing 2.7 Showing a little data using the `.show()` method.

```
book.show()

# +--------------------+
# |               value|
# +--------------------+
# |The Project Guten...|
# |                    |
# |This eBook is for...|
# |almost no restric...|
# |re-use it under t...|
# |with this eBook o...|
# |                    |
# |                    |
# |Title: Pride and ...|
# |                    |
# | Author: Jane Austen|
# |                    |
# |Posting Date: Aug...|
# |Release Date: Jun...|
# |Last Updated: Mar...|
# |                    |
# |   Language: English|
# |                    |
# |Character set enc...|
# |                    |
# +--------------------+
# only showing top 20 rows
```

The `show()` method takes three optional parameters.

- `n` can be set to any positive integer, and will display that number of rows.
- `truncate`, if set to true, will truncate the columns to display only 20 characters. Set to `False` to display the whole length, or any positive integer to truncate to a specific number of characters.
- `vertical` takes a Boolean value and, when set to `True`, will display each record as a small table. Try it!

The code in shows a couple options, stating with showing 10 records and truncating then at 50

characters. We can see more of the text now!

Listing 2.8 Showing less length, more width with the `show()` method parameters

```
book.show(10, truncate=50)

# +--------------------------------------------------+
# |                                             value|
# +--------------------------------------------------+
# |The Project Gutenberg EBook of Pride and Prejud...|
# |                                                  |
# |This eBook is for the use of anyone anywhere at...|
# |almost no restrictions whatsoever.  You may cop...|
# |re-use it under the terms of the Project Gutenb...|
# |    with this eBook or online at www.gutenberg.org|
# |                                                  |
# |                                                  |
# |                       Title: Pride and Prejudice|
# |                                                  |
# +--------------------------------------------------+
# only showing top 10 rows
```

With the `show()` and `printSchema()` methods under your belt, you're now fully ready to experiment with your data.

SIDEBAR **Non-lazy Spark?**

If you are coming from an other data frame implementation, such as Pandas or R `data.frame`, you might find it odd to see the structure of the data frame instead of a summary of the data when calling the variable. The `show()` method might appear as a nuisance to you.

If we take a step back and think about PySpark's use-cases, it makes a lot of sense. `show()` is an action, since it perform the visible work of printing data on the screen. As savvy PySpark programmers, we want to avoid to accidentally trigger the chain of computations, so the Spark developers made `show()` explicit. When building a complicated chain of transformations, triggering its execution is a lot more annoying and time-consuming than having to type the `show()` method when you're ready.

That being said, there are some moments, especially when learning, when you want your data frames to be evaluated after each transformation (which we call *eager evaluation*). Since Spark 2.4.0, you can configure the `SparkSession` object to support printing to screen. We will cover how to create a `SparkSession` object in greater details in Chapter 3, but if you want to use eager evaluation in the shell, you can paste the following code in your shell.

```
from pyspark.sql import SparkSession

spark = (SparkSession.builder
                     .config("spark.sql.repl.eagerEval.enabled", "True")
                     .getOrCreate())
```

All the examples in the book assume that the data frames are evaluated lazily, but this option can be useful if you're demonstrating Spark. Use it as you see fit, but remember that Spark owe a lot of its performance to its lazy evaluation. You'll be leaving some extra horsepower on the table!

Our data is ingested and we've been able to see the two important aspects of our data frame:

- its structure, via the `printSchema()` method;
- a subset of the data it contains, via the `show()` method.

We can now start the real work: performing transformations on the data frame to accomplish our goal. Let's take some time to review our 5 steps we outlined at the beginning of the chapter.

1. **[DONE]** *Read*: Read the input data (we're assuming a plain text file)
2. *Token*: Tokenize each word
3. *Clean*: Remove any punctuation and/or tokens that aren't words.
4. *Count*: Count the frequency of each word present in the text
5. *Answer:* Return the top 10 (or 20, 50, 100)

Our next step will be to tokenize or separate each word so we can clean and count them.

2.5 Moving from a sentence to a list of words

When ingesting our selected text into a data frame, PySpark created one record for each line of text, and provided a `value` column of type String. In order to tokenize each word, we need to split each string into a list of distinct words.

I'll start by providing the code in one fell swoop, and then we'll break down each step one at a time. You can see it in all its glory in .

Listing 2.9 Splitting our lines of text into arrays or words

```
from pyspark.sql.functions import split

lines = book.select(split(book.value, " ").alias("line"))

lines.show(5)

# +--------------------+
# |               value|
# +--------------------+
# |[The, Project, Gu...|
# |                  []|
# |[This, eBook, is,...|
# |[almost, no, rest...|
# |[re-use, it, unde...|
# +--------------------+
# only showing top 5 rows
```

In a single line of code (I don't count the import or the `show()` which is only being used to display the result), we've done quite a lot. The remainder of this section will introduce basic column operations and explain how we can build our tokenization step as a one-liner. More specifically, we'll learn about

- The `select()` method and its canonical usage, which is selecting data.
- The `alias()` method to rename transformed columns
- Importing column functions from `pyspark.sql.functions` and using them.

2.5.1 Selecting specific columns using `select()`

This section will introduce the most basic functionality of `select()`, which is to select one or more columns from your data frame. It's a conceptually very simple method, but provides the foundation for many additional operations on your data.

In PySpark's world, a data frame is made out of `Column` objects, and you perform transformations on them. The most basic transformation is the identity, where you return exactly what was provided to you. If you've used SQL in the past, you might think that this sounds like a "SELECT" statement, and you'd be right! You also get a free pass: the method name is also conveniently named `select()`.

We'll go over a quick example: selecting the only columns of our `book` data frame. Since we already know the expected output, we can focus on the gymnastics fo the `select()` method. provides the code performing that very useful task.

Listing 2.10 The simplest select statement ever, provided by PySpark

```
book.select(book.value)
```

PySpark provides for each column in its data frame a dot notation that refers to the column. This is definitely the simplest way to select a column, as long as the name doesn't contain any funny characters: PySpark will accept `$!@#` as a column name, but you won't be able to use the dot notation for this column.

PySpark provides more than one way to select columns. I displayed the three most common in .

Listing 2.11 Selecting the `value` column from the `book` data frame, three ways

```
from pyspark.sql.functions import col

book.select(book.value)
book.select(book["value"])
book.select(col("value"))
```

The first one is our old trusty dot notation that we got acquainted with a few paragraphs ago.

The second one uses brackets instead of the dot to name the column. It addresses the `$!@#` problem since you pass the name of the column as a string.

The last one uses the `col` function from the `pyspark.sql.functions` module. The main difference here is that you don't specify that the column comes from the `book` data frame. This will become very useful when working with more complex data pipelines in Part 2 of the book. I'll use the `col` object as much as I can since I consider its usage to be more idiomatic and it'll prepare us for more complex use-cases.

2.5.2 Transforming columns: splitting a string into a list of words

We just saw a very simple way to select a column in PySpark. We will now build on this foundation by selecting a transformation of a column instead. This provides a powerful and flexible way to express our transformations, and as you'll see, this pattern will be frequently used when manipulating data.

PySpark provides a `split()` function in the `pyspark.sql.functions` module for splitting a longer string into a list of shorter strings. The most popular use-case for this function is to split a sentence into words. The `split()` function takes two parameters.

1. A column object containing `strings`

2. A Java regular expression delimiter to split the strings against.

Since we want to split words, we won't over-complicate our regular expression and just use the space character to split. shows the results of our code.

Listing 2.12 Splitting our lines of text into lists of words

```
from pyspark.sql.functions import col, split

lines = book.select(split(col("value"), " "))

lines

# DataFrame[split(value,  ): array<string>]

lines.printSchema()

# root
#  |-- split(value,  ): array (nullable = true)
#  |    |-- element: string (containsNull = true)

lines.show(5)

# +--------------------+
# |     split(value,  )|
# +--------------------+
# |[The, Project, Gu...|
# |                  []|
# |[This, eBook, is,...|
# |[almost, no, rest...|
# |[re-use, it, unde...|
# +--------------------+
# only showing top 5 rows
```

The `split` functions transformed our `string` column into an `array` column, containing one or more `string` elements. This is what we were expecting: even before looking at the data, seeing that the structure behaves according to plan is a good way to sanity-check our code.

Looking at the 5 rows we've printed, we can see that our values are now separated by a comma and wrapped in square brackets, which is how PySpark visually represents an array. The second record is empty, so we just see `[]`, an empty array.

PySpark's built-in functions for data manipulations are extremely useful and you should definitely spend a little bit of time going over the API documentation to see what's available there. If you don't find exactly what you're after, Chapter 6 will cover how you can create your own function over `Column` objects.

SIDEBAR **Advanced topic: PySpark's architecture and the JVM heritage**

If you're like me, you might be interested to see how PySpark builds its core `pyspark.sql.functions` functions. If you look at the source code for `split()`, you might be in for a disappointment.

```
since(1.5)
@ignore_unicode_prefix
def split(str, pattern):
    """
    Splits str around pattern (pattern is a regular expression).

    .. note:: pattern is a string represent the regular expression.

    >>> df = spark.createDataFrame([('ab12cd',)], ['s',])
    >>> df.select(split(df.s, '[0-9]+').alias('s')).collect()
    [Row(s=[u'ab', u'cd'])]
    """
    sc = SparkContext._active_spark_context
    return Column(sc._jvm.functions.split(_to_java_column(str), pattern))
```

It effectively refers to the `split` function of the `sc_jvm.functions` object. This has to do with how the data frame was built. PySpark's uses a translation layer to call JVM functions for its core functions. This makes PySpark faster, since you're not transforming your Python code into JVM one all the time: it's already done for you. It also makes porting PySpark to another platform a little easier: if you can call the JVM functions directly, you don't have to re-implement everything.

This is one of the trade-offs of standing on the shoulders of the Spark giant. This also explains why PySpark uses JVM-base regular expressions instead of the Python ones in its built-in functions. Part 3 will expand on this subject greatly, but in the meantime, don't be surprised if you explore PySpark's source code!

PySpark renamed our column in a very weird way: `split(value, )` isn't what I'd consider an awesome name for our column. Just like the infomercials say, *there must be a better way!*.

2.5.3 Renaming columns: `alias` and `withColumnRenamed`

When performing transformation on your columns, PySpark will give a default name to the resulting column. In our case, we were blessed by the `split(value, )` name after splitting our value column using a space as the delimiter. While accurate, it's definitely not programmer friendly.

There is an implicit assumption that you'll want to rename the resulting column yourself, using the `alias()` method. It's usage isn't very complicated: when applied to a column, it takes a single parameter, and returns the column it was applied to, with the new name. A simple demonstration is provided in .

Listing 2.13 Our data frame before and after the aliasing

```
book.select(split(col("value"), " ")).printSchema()
# root
#  |-- split(value,  ): array (nullable = true)   ❶
#  |    |-- element: string (containsNull = true)

book.select(split(col("value"), " ").alias("line")).printSchema()

# root
#  |-- line: array (nullable = true)   ❷
#  |    |-- element: string (containsNull = true)
```

❶ Our new column is called `split(value, )`, which isn't really pretty

❷ We aliased our column to the name `line`. Much better!

`alias()` provides a clean and explicit way to name your columns after you've performed work on it. On the other hand, it's not the only renaming player in town. Another equally valid way to do so is by using the `.withColumnRenamed()` method on the data frame. It takes two parameters: the current name of the column and the wanted name of the column. Since we're already performing work on the column with `split`, chaining `alias` makes a lot more sense than using another method. shows you the two different approaches.

When writing your own code, choosing between those two options is pretty easy:

- when you're using a method where you're specifying which columns you want to appear (like `select` in our case here, but the next chapters will have many other examples), use `alias`.
- if you just want to rename a column without changing the rest of the data frame, use `.withColumnRenamed`.

Listing 2.14 Renaming a column via `alias` on the column and `withColumnRenamed` on the DataFrame

```
# This looks a lot cleaner
lines = book.select(split(book.value, " ").alias("line"))

# This is messier, and you have to remember the name PySpark assigns automatically
lines = book.select(split(book.value, " "))
lines = lines.withColumnRenamed("split(value,  )", "line")
```

This section introduced a new set of PySpark fundamentals: we learned how to select not only plain columns, but also column transformations. We also learned how to explicitly name the resulting columns, avoiding PySpark's predictable but jarring naming convention. We can then move forward with the remainder of the operations. If we look at our 5 steps, we're halfway done with step 2: we have a list of words, but we need for each token or word to be its own records.

1. **[DONE]** *Read*: Read the input data (we're assuming a plain text file)
2. **[IN PROGRESS]** *Token*: Tokenize each word

3. *Clean*: Remove any punctuation and/or tokens that aren't words.
4. *Count*: Count the frequency of each word present in the text
5. *Answer:* Return the top 10 (or 20, 50, 100)

2.6 Reshaping your data: exploding a list into rows

When working with data, a key element in data preparation is making sure that it "fits the mold": this means making sure that the structure containing the data is logical and appropriate for the work at hand. At the moment, each record of our data frame contains multiple words into an array of strings. It would be better to have one record for each word.

Enter the `explode` function. When applied to a column containing a container-like data structure (such as an array), it'll take each element and give it its own row. This is much more easier explained visually than using words, so explains the process.

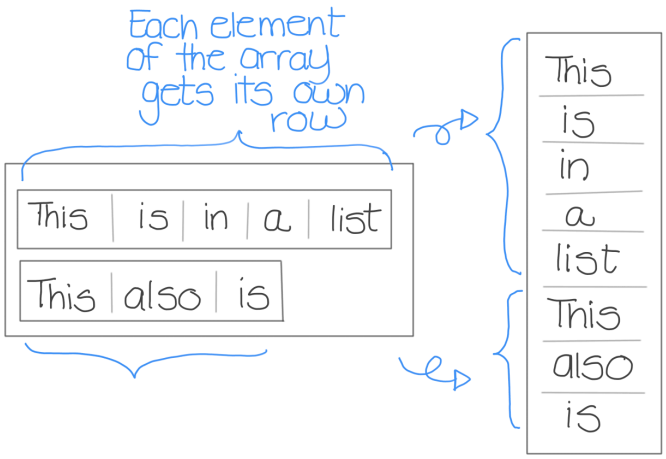


Figure 2.4 Exploding a data frame of array[String] into a data frame of String. Each element of each array becomes its own record.

The code follows the same structure as split, and you can see the results in . We now have a data frame containing at most one word per row. We are almost there!

Before continuing our data processing journey, we can take a step back and look at a sample of the data. Just by looking at the 15 rows returned, we can see that `Prejudice,` has a comma and that the cell between `Austen` and `This` contains the empty string. That gives us a good blueprint of the next steps that needs to be performed before we start analyzing word frequency.

Listing 2.15 Exploding a column of arrays into rows of elements

```
from pyspark.sql.functions import explode, col

words = lines.select(explode(col("line")).alias("word"))

words.show(15)
# +----------+
# |      word|
# +----------+
# |       The|
# |   Project|
# | Gutenberg|
# |     EBook|
# |        of|
# |     Pride|
# |       and|
# |Prejudice,|
# |        by|
# |      Jane|
# |    Austen|
# |          |
# |      This|
# |     eBook|
# |        is|
# +----------+
# only showing top 15 rows
```

Looking back at our 5 steps, we can now conclude step 2, and our words are tokenized. Let's attack the third one, where we'll be cleaning our words to simplify the counting.

1. **[DONE]** *Read*: Read the input data (we're assuming a plain text file)
2. **[DONE]** *Token*: Tokenize each word
3. *Clean*: Remove any punctuation and/or tokens that aren't words.
4. *Count*: Count the frequency of each word present in the text
5. *Answer:* Return the top 10 (or 20, 50, 100)

2.7 Working with words: changing case and removing punctuation

So far, with `split` and `explode`, our pattern has been the following: find the relevant function in `pyspark.sql.functions`, apply it, profit! This section will use the same winning formula to normalize the case of our words and remove punctuation, so we'll walk a little faster.

contains the source code to lower the case of all the words in the data frame. The code should look very familiar: we select a column transformed by `lower`, a PySpark function lowering the case of the data inside the column passed as a parameter. We then alias the resulting column to `word` to avoid PySpark's default nomenclature. Illustrated, it could look approximately like .

Listing 2.16 Lower the case of the words in the data frame

```
from pyspark.sql.functions import lower
words_lower = words.select(lower(col("word")).alias("word_lower"))

words_lower.show()

# +-----------+
# | word_lower|
# +-----------+
# |        the|
# |    project|
# |  gutenberg|
# |      ebook|
# |         of|
# |      pride|
# |        and|
# | prejudice,|
# |         by|
# |       jane|
# |     austen|
# |           |
# |       this|
# |      ebook|
# |         is|
# |        for|
# |        the|
# |        use|
# |         of|
# |     anyone|
# +-----------+
# only showing top 20 rows
```

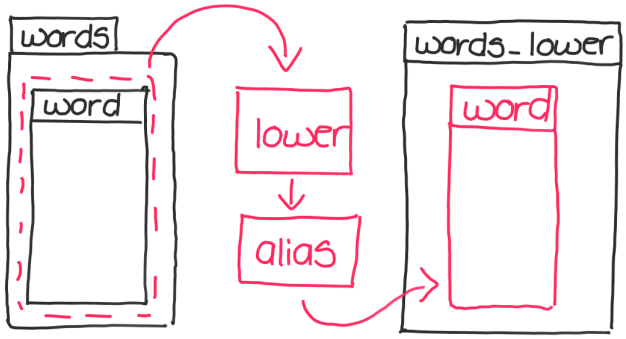


Figure 2.5 Applying lower to a column and aliasing the resulting column: a creative explanation

Removing punctuation and other non-useful characters can be a little trickier. We won't improvise a full NLP (Natural Language Processing, which is an amazingly field of data analysis/science that focuses on text. We'll cover it briefly in Chapter 9!) library here, relying on the functionality PySpark provides in its data manipulation toolbox. In the spirit of keeping this exercise simple, we'll keep the first contiguous group of letters as the word, including apostrophes, and remove the rest. It will effectively remove punctuation, quotation marks and other symbols, at the expense of being less robust with more exotic word construction. shows the code in all its splendour.

SIDEBAR **Regular expressions for the rest of us**

PySpark uses regular expressions in two functions we used so far: `regexp_extract()` and `split()`. You do not have to be a regexp expert to work with PySpark (I certainly am not). Through the book, each time that I'll use a non-trivial regular expression, I'll provide a plain English definition so you can follow along.

If you are interested in building your own, the RegExr (https://regexr.com/) website is really useful, as well as the *Regular Expression Cookbook*, by Steven Levithan and Jan Goyvaerts (O'Reilly, 2012).

Listing 2.17 Using `regexp_extract` to keep what looks like a word

```
from pyspark.sql.functions import regexp_extract
words_clean = words_lower.select(
    regexp_extract(col("word_lower"), "[a-z]*", 0).alias("word")    ❶
)

words_clean.show()

# +---------+
# |     word|
# +---------+
# |      the|
# |  project|
# |gutenberg|
# |    ebook|
# |       of|
# |    pride|
# |      and|
# |prejudice|
# |       by|
# |     jane|
# |   austen|
# |         |
# |     this|
# |    ebook|
# |       is|
# |      for|
# |      the|
# |      use|
# |       of|
# |   anyone|
# +---------+
# only showing top 20 rows
```

❶ We only match for multiple lower-case characters (between a and z). The star (*) will match for 0 or more occurrences.

Our data frame of words looks pretty regular by now, with the exception of the empty cell between `austen` and `this`. We will solve this with a judicious usage of filtering.

2.8 Filtering rows

An important data manipulation operation is to be able to filter records according to a certain predicate. In our case, blank cells shouldn't be considered: they're not words! Conceptually, we should be able to provide a test to perform on each records: if it returns true, we keep the record. False? You're out!

PySpark provides not one, but two identical methods to perform this task. You can either use `.filter()` or its alias `.where()`. This duplication is to ease the transition for users coming from other data processing engines or libraries: some use one, some the other. PySpark provides both, so no arguments possible! I personally prefer `where()` because it's one character less and my w key is less used than f, but you might have other motives. If we look at , we can see that columns can be compared to values using the usual Python comparison operators. In this case, we're using the "not equal", or `!=`.

Listing 2.18 Filtering rows in your data frame, using `where` or `filter`.

```
words_nonull = words_clean.where(col("word") != "")

words_nonull.show()

# +---------+
# |     word|
# +---------+
# |      the|
# |  project|
# |gutenberg|
# |    ebook|
# |       of|
# |    pride|
# |      and|
# |prejudice|
# |       by|
# |     jane|
# |   austen|
# |     this| <-- See, the blank cell is gone!
# |    ebook|
# |       is|
# |      for|
# |      the|
# |      use|
# |       of|
# |   anyone|
# | anywhere|
# +---------+
# only showing top 20 rows
```

We could have tried to filter earlier in our program. It's a trade-off to consider: if we filtered too early, our filtering clause would have been comically complex for no good reason. Since PySpark caches all the transformations until an action is triggered, we can focus on the readability of our code and let Spark optimize our intent, like we saw in Chapter 1. We'll see in Chapter 3 how you can transform PySpark code so it almost reads like a series of written instructions and take advantage of the lazy evaluation.

This seems like a good time to take a break and reflect on what we accomplished so far. If we look at our 5 steps, we're 60% of the way there. Our cleaning step took care of non-letter characters and filtered the empty records. We're ready for counting and displaying the results of our analysis.

1. **[DONE]** *Read*: Read the input data (we're assuming a plain text file)
2. **[DONE]** *Token*: Tokenize each word
3. **[DONE]** *Clean*: Remove any punctuation and/or tokens that aren't words.
4. *Count*: Count the frequency of each word present in the text
5. *Answer:* Return the top 10 (or 20, 50, 100)

In terms of PySpark operations, we covered a huge amount of ground in the data manipulation space. You can now select not only columns but transformations of columns, renaming them as you please after the fact. We learned how to break nested structures, such as arrays, into single records. We finally learn how to filter records using simple tests.

We can now rest. The next chapter will cover the end of our program. We will also be looking at bringing our code in one single file, moving away from the REPL into batch mode. We'll explore options to simplify and increase the readability of our program, and then finish by scaling it to larger corpus of texts.

2.9 Summary

- Almost all PySpark programs will revolve around 3 major steps: reading, transforming and exporting data.
- PySpark provides a REPL (read, eval, print loop) via the `pyspark` shell where you can experiment interactively with data.
- A PySpark's data frame is a collection of columns. You operate on the structure using chained transformations. PySpark will optimize the transformations and perform the work only when you submit an action, such as `show()`. This is one of the pillars of PySpark's performance.
- PySpark's repertoire of functions that operate on columns are located in `pyspark.sql.functions`.
- You can select columns or transformed columns via the `select()` statement.
- You can filter columns using the `where()` or `filter()` methods and providing a test that will return `True` or `False`, only the records returning `True` will be kept.
- PySpark can have columns of nested values, like arrays of elements. In order to extract the elements into distinct records, you need to use the `explode()` method.

2.10 Exercises

2.10.1 Exercise 2.1

Rewrite the following code snippet, removing the `withColumnRenamed` method. Which version is clearer and easier to read?

```
from pyspark.sql.functions import col, length

# The `length` function returns the number of characters in a string column.

ex21 = (
    spark.read.text("./data/Ch02/1342-0.txt")
    .select(length(col("value")))
    .withColumnRenamed("length(value)", "number_of_char")
)
```

2.10.2 Exercice 2.2

The following code blocks gives an error. What is the problem and how can you solve it?

```
from pyspark.sql.functions import col, greatest

ex22.printSchema()
# root
#  |-- key: string (containsNull = true)
#  |-- value1: long (containsNull = true)
#  |-- value2: long (containsNull = true)

# `greatest` will return the greatest value of the list of column names,
# skipping null value

# The following statement will return an error
ex22.select(
    greatest(col("value1"), col("value2")).alias("maximum_value")
).select(
    "key", "max_value"
)
```

2.10.3 Exercise 2.3

Let's take our `words_nonull` data frame, available in . You can use the code in the repository (`code/Ch02/end_of_chapter.py`) into your REPL to get the data frame loaded.

a) Remove all of the occurrences of the word "is"

b) (Challenge) Using the `length` function explained in exercise 2.1, keep only the words with more than 3 characters.

2.10.4 Exercise 2.4

The `where` clause takes a Boolean expression over one or many column to filter the data frame (see). Beyond the usual Boolean operators (>, <, ==, , >=, !=), PySpark provides other functions returning Boolean columns in the `pyspark.sql.functions` module.

A good example is the `isin()` function, which takes a list of values as a parameter, and will return only the records where the value in the column equals a member of the list.

Let's say you want to *remove* the words `is`, `not`, `the` and `if` from your list of words, using a single `where()` method on the `words_nonull` data frame (see exercise 2.3). Write the code to do so.

2.10.5 Exercise 2.5

One of your friends come to you with the following code. They have no idea why it doesn't work. Can you diagnose the problem, explain why it is an error and provide a fix?

```
from pyspark.sql.functions import col, split

book = spark.read.text("./data/ch02/1342-0.txt")

book = book.printSchema()

lines = book.select(split(book.value, " ").alias("line"))

words = lines.select(explode(col("line")).alias("word"))
```

3

Submitting and scaling your first PySpark program

This chapter covers:

- Summarizing data using `groupby` and a simple aggregate function
- Ordering results for display
- Writing data from a data frame
- Using `spark-submit` to launch your program in batch mode
- Simplify the writing of your PySpark using method chaining
- Scaling your program almost for free!

Chapter 2 dealt with all the data preparation work for our word frequency program. We *read* the input data, *tokenized* each word and *cleaned* our records to only keep lower-case words. If we bring out our outline, we only have Step 4 and 5 to complete.

1. **[DONE]** *Read*: Read the input data (we're assuming a plain text file)
2. **[DONE]** *Token*: Tokenize each word
3. **[DONE]** *Clean*: Remove any punctuation and/or tokens that aren't words.
4. *Count*: Count the frequency of each word present in the text
5. *Answer:* Return the top 10 (or 20, 50, 100)

After tackling those two last steps, we'll look at packaging our code in a single file to be able to submit it to Spark without having to launch a shell. We'll take a look at our completed program and look at simplifying our program by removing intermediate variables. We'll finish with scaling our program to accommodate more data sources.

3.1 Grouping records: Counting word frequencies

If you take our data frame in the same shape as it was left at the end of Chapter 2 (hint: look at `code/Ch02/end_of_chapter.py` if you want to catch up), there is not much to be done. Having a data frame containing one single word per record, we just have to count the word occurrences and take the top contenders. This section will show how to count records using the `GroupedData` object and perform an aggregation function (here counting the items) on each group. A more general blueprint for grouping and aggregating data will be touched upon in Chapter 4, but we'll see the basics in this Chapter.

The easiest way to count record occurrence is to use the `groupby` method, passing the columns we wish to group on as a parameter. The code in shows that the returned value is a `GroupedData` object, not a `DataFrame`. I call this `GroupedData` object a *transitional object*: PySpark grouped our data frame on the `word` column, waiting for instructions on how to summarize the information contained in each group. Once we apply the `count()` method, we get back a data frame containing the grouping column `word`, as well as `count` column containing the number of occurrences for each word. A visual interpretation of how a `DataFrame` morphes into a `GroupedData` object is on display in

Listing 3.1 Counting word frequencies using `groupby()` and `count()`

```
groups = words_nonull.groupby(col("word"))

groups

# <pyspark.sql.group.GroupedData at 0x10ed23da0>

results = words_nonull.groupby(col("word")).count()

results

# DataFrame[word: string, count: bigint]

results.show()

# +-------------+-----+
# |         word|count|
# +-------------+-----+
# |       online|    4|
# |         some|  203|
# |        still|   72|
# |          few|   72|
# |         hope|  122|
# |        those|   60|
# |     cautious|    4|
# |       lady's|    8|
# |    imitation|    1|
# |          art|    3|
# |      solaced|    1|
# |       poetry|    2|
# |    arguments|    5|
# | premeditated|    1|
# |      elevate|    1|
# |       doubts|    2|
# |    destitute|    1|
# |    solemnity|    5|
# |gratification|    1|
# |    connected|   14|
# +-------------+-----+
# only showing top 20 rows
```

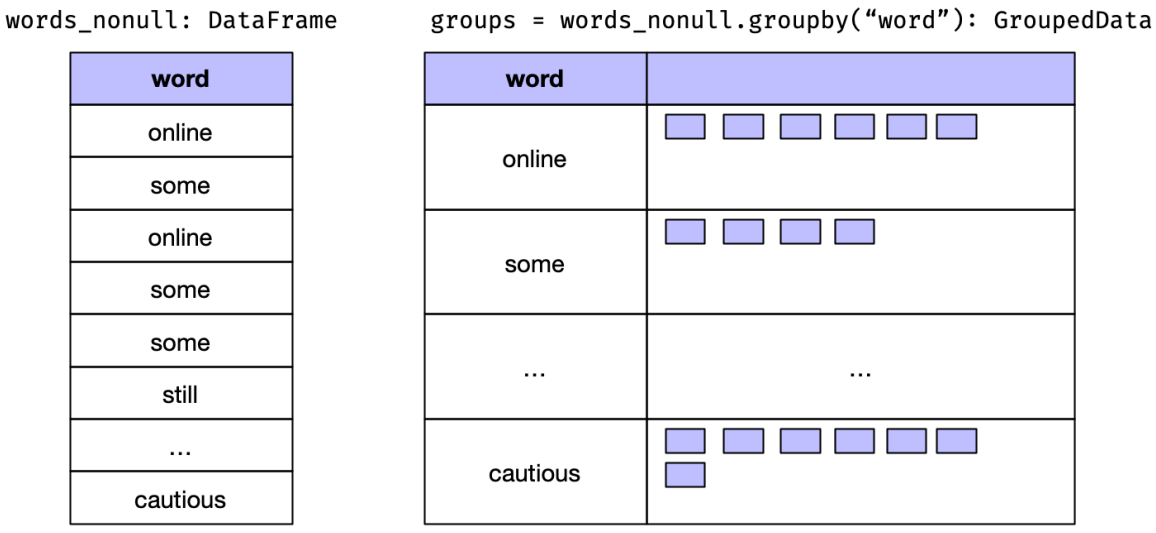


Figure 3.1 A schematic representation of our groups object. Each small box represents a record.

Peeking at the `results` data frame in , we see that the results are in no specific order. As a matter of fact, I'd be very surprised if you had the exact same order of words as me! This has to do with how PySpark manages data: in Chapter 1, we learned that PySpark distributes the data across multiple nodes. When performing a grouping function, such as `groupby`, each worker performs the work on its assigned data. `groupby` and `count` are transformations, so PySpark will queue them lazily until we request an action. When we pass the `show` method to our results data frame, it triggers the chain of computation that we see in .

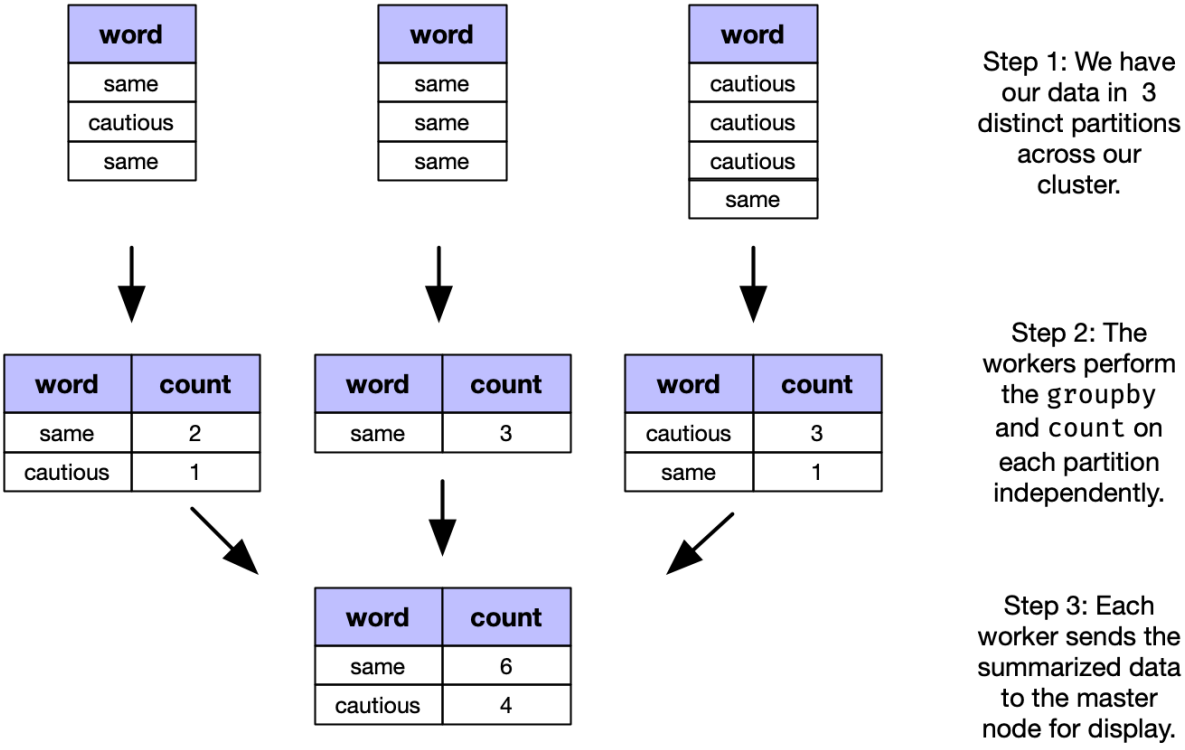


Figure 3.2 A distributed group by on our words_nonull data frame. The work is performed in a distributed fashion until we need to assemble the results in a cohesive display, via show().

Because of the distributed and lazy nature of PySpark, it makes sense to not care about the ordering of records until explicitly mentioned. Since we wish to see the top words on display, let's put a little order in our data frame and, by the same occasion, complete the last step of our program.

3.2 Ordering the results on the screen using `orderBy`

In , we explained why PySpark doesn't necessarily maintain order of records when performing transformations. If we look at our 5 step blueprint, the last step is to return the top N records, for different values of N. We already know how to show a specific number of records, so this section will focus on ordering the records in a data frame.

1. **[DONE]** *Read*: Read the input data (we're assuming a plain text file)

2. **[DONE]** *Token*: Tokenize each word
3. **[DONE]** *Clean*: Remove any punctuation and/or tokens that aren't words.
4. **[DONE]** *Count*: Count the frequency of each word present in the text
5. *Answer:* Return the top 10 (or 20, 50, 100)

The method PySpark provides for ordering records in a data frame is `orderBy`. Without further ado, let's take a look at , which performs and shows the results of this computation.

Listing 3.2 Displaying the top 10 words in Jane's Austen *Pride and Prejudice*

```
results.orderBy("count", ascending=False).show(10)

# +----+-----+
# |word|count|
# +----+-----+
# | the| 4480|
# |  to| 4218|
# |  of| 3711|
# | and| 3504|
# | her| 2199|
# |   a| 1982|
# |  in| 1909|
# | was| 1838|
# |   i| 1749|
# | she| 1668|
# +----+-----+
# only showing top 10 rows
```

The list is very much unsurprising: even though we can't argue with Ms. Austen's vocabulary, she isn't immune to the fact that the English language needs pronouns and other very common words. In natural language processing, those words are called *stop words* and would be removed. As far as we are concerned, we solved our original query and can rest easy. Should you want to get the top 20, top 50, or even top 1,000, it's easily done by changing the parameter to `show()`.

SIDEBAR **PySpark's method naming convention zoo**

If you have a very good sense of details, you might have noticed that we used `groupby` (lowercase), but `orderBy` (lowerCamelCase, where you capitalize each word but the first). This seems like an odd design choice.

`groupby` is in fact an alias for `groupBy`, just like `where` is an alias of `filter`. My guess is that the PySpark developers found that a lot of typing mistakes were avoided by accepting the two cases. `orderBy` didn't have that luxury, for a reason that escape my understanding, so we need to be mindful. You can see the output of IPython's auto-complete for those two methods in .

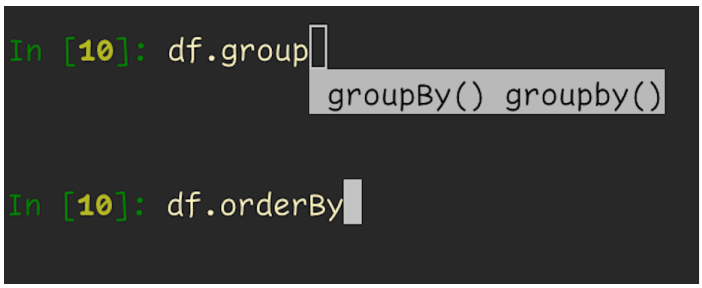


Figure 3.3 PySpark's camelcase vs camelCase

SIDEBAR Part of this incoherence is due to Spark's heritage. Scala prefers camelCase for methods. On the other hand, we saw `regexp_extract`, which uses Python's preferred snake_case (words separated by an underscore) in Chapter 2. There is no magic secret here: you'll have to be mindful about the different case conventions at play in PySpark.

Showing results on the screen is great for quick assessment, but most of the time, you'll want them to have some sort of longevity. It's much better to save those results into a file, so we'll be able to re-use those results without having to compute everything each time.

3.3 Writing data from a data frame

Having the data on the screen is great for interactive development, but you'll often want to export your results. PySpark treats writing data a little differently than most data processing libraries, since it can scale to immense volumes of data. For this, we'll start by naively write our results in a CSV file, and see how PySpark performs the job.

shows the code and results. A data frame exposes the `write` method, which we can specialize for CSV by chaining the `csv` method. This is very consistent with the `read` method we saw in Chapter 2. If we look at the results, we can see that PySpark didn't create a `results.csv` file. Instead, it created a directory of the same name, and put 201 files inside the directory (200 CSVs + 1 `_SUCCESS` file).

Listing 3.3 Writing our results in multiple CSV files, one per partition

```
results.write.csv('./results.csv')

# The following command is run using the shell.
# In IPython, you can use the bang pattern (! ls -l)
# to get the same results without leaving the console.

#`ls -l` is a Unix command listing the content of a directory.
# On windows, you can use `dir` instead

$ ls -l
# [...]
# -rw-r--r--@   1 jonathan_rioux  247087069  724726 Jul 30 17:30 1342-0.txt
# drwxr-xr-x  404 jonathan_rioux  247087069   12928 Aug  4 13:31 results.csv   ❶

$ ls -l results.csv/
# [...]
# -rw-r--r--  1 jonathan_rioux  247087069    0 Aug
  4 13:31 _SUCCESS   ❷
# -rw-r--r--  1 jonathan_rioux  247087069  468 Aug
  4 13:31 part-00000-615b75e4-ebf5-44a0-b337-405fccd11d0c-c000.csv
# [...]
# -rw-r--r--  1 jonathan_rioux  247087069  353 Aug
  4 13:31 part-00199-615b75e4-ebf5-44a0-b337-405fccd11d0c-c000.csv   ❸
```

❶ The results are written in a directory called `results.csv`

❷ The `_SUCCESS` file means the operation was successful

❸ We have `part-00000` to `part-00199`, which means our results are split across 200 files

There it is, ladies and gentleman! The first moment where we have to care about PySpark's distributed nature.

Just like PySpark will distribute the transformation work across multiple workers, it'll do the same for writing data. While it might look like a nuisance for our simple program, it is tremendously useful when working in distributed environments. When you have a large cluster of nodes, having many smaller files makes it easy to logically distribute reading and writing the data, making it way faster than having a single massive file.

By default, PySpark will give you 1 file per partition. This means that our program, as run on my machine, yields 200 partitions at the end. This isn't the best for portability. In order to reduce the number to partitions, we can apply the `coalesce` method with the desired number of partitions. shows the difference of using `coalesce(1)` on our data frame before writing to disk. We still get a directory, but there is a single CSV file inside of it. Mission accomplished!

Listing 3.4 Writing our results under a single partition

```
results.coalesce(1).write.csv('./results_single_partition.csv')

$ ls -l
# [...]
# -rw-r--r--@   1 jonathan_rioux  247087069  724726 Jul 30 17:30 1342-0.txt
# drwxr-xr-x  404 jonathan_rioux  247087069   12928 Aug  4 13:31 results.csv
# drwxr-xr-x    6 jonathan_rioux  247087069     192 Aug  4 13:43 results_single_partition.csv

$ ls -l results_single_partition.csv/
# [...]
# -rw-r--r--  1 jonathan_rioux  247087069      0 Aug
  4 13:43 _SUCCESS
# -rw-r--r--  1 jonathan_rioux  247087069  70993 Aug
  4 13:43 part-00000-f8c4c13e-a4ee-4900-ac76-de3d56e5f091-c000.csv
```

NOTE You might have realized that we're not ordering the file before writing it. Since our data here is pretty small, we could have written the words by decreasing order of frequency. If you have a very large data set, this operation will be quite expensive. Furthermore, since reading is a potentially distributed operation, what guarantees that it'll get read the exact same way? Never assume that your data frame will keep the same ordering of records unless you explicitly ask via `orderBy()`.

Our workflow has been pretty interactive so far. We write one or two lines of text before showing the result to the terminal. As we get more and more confident with operating on the data frame's structure, those shows will become fewer.

Now that we've performed all the necessary steps interactively, let's look in putting our program in a single file and looking at refactoring opportunities.

3.4 Putting it all together: counting

Interactive development is fantastic for rapid iteration of our code. When developing programs, it's great to experiment and validate our thoughts through rapid code inputs into a shell. When the experimentation is over, it's good to bring our program into a cohesive body of code.

The `pyspark` shell allows to go back in history using the directional arrows of your keyboard, just like a regular python REPL. To make things a bit easier, I am providing the step by step program in . This section is dedicated to streamline and make our code terser and more readable.

Listing 3.5 Our first PySpark program, dubbed "Counting Jane Austen"

```
from pyspark.sql.functions import col, explode, lower, regexp_extract, split

book = spark.read.text("./data/ch02/1342-0.txt")

lines = book.select(split(book.value, " ").alias("line"))

words = lines.select(explode(col("line")).alias("word"))

words_lower = words.select(lower(col("word")).alias("word"))

words_clean = words_lower.select(
    regexp_extract(col("word"), "[a-z']*", 0).alias("word")
)

words_nonull = words_clean.where(col("word") != "")

results = words_nonull.groupby(col("word")).count()

results.orderBy("count", ascending=False).show(10)

results.coalesce(1).write.csv("./results_single_partition.csv")
```

This program runs perfectly id you paste its entirety in the `pyspark` shell. With everything in the same file, we can look at making our code friendlier and easier for future you to come back at it.

3.4.1 Simplifying your dependencies with PySpark's import conventions

This program uses five distinct functions from the `pyspark.sql.functions` modules. We should probably replace this with a qualified import, which is Python's way to import a module by assigning a keyword to it. While there is no hard rule, the common wisdom is to use `F` to refer to PySpark's functions. shows the before and after.

Listing 3.6 Simplifying our PySpark functions import

```
# Before
from pyspark.sql.functions import col, explode, lower, regexp_extract, split

# After
import pyspark.sql.functions as F
```

Since `col`, `explode`, `lower`, `regexp_extract` and `split` are all in the `pyspark.sql.functions`, we can import the whole module. Since the new import statement imports the entirety of the `pyspark.sql.functions` module, we assign the keyword (or key-letter) `F`. The PySpark community seems to have implicitly settled on using `F` for `pyspark.sql.functions` and I encourage you to do the same. It'll make your programs consistent, and since many functions in the module share their name with Pandas or Python built-in functions, you'll avoid name clashes.

WARNING It can be very tempting to do a start import like `from pyspark.sql.functions import *`. Do not fall into that trap! It'll make it hard for your readers which functions comes from PySpark and which comes from regular Python. In Chapter 8, when we'll use user defined functions (UDF), this separation will become even more important. Good coding hygiene rules!

That was easy enough. Let's look at how we can simplify our program flow by using one of my favourite aspect of PySpark, its *chaining* abilities.

3.4.2 Simplifying our program via method chaining

If we look at the transformation methods we applied on our data frames (`select()`, `where()`, `groupBy()` and `count()`), they all have something in common: they take a structure as a parameter— the data frame or `GroupedData` in the case of `count()`— and return a structure. There is no concept of in-place modification in PySpark: all transformations can be seen as a pipe that ingests a structure and returns a modified structure. This section will look at what is probably my favourite aspect of PySpark: method chaining.

Our program uses intermediate variables quite a lot: everytime we perform a transformation, we assigned the result to a new variable. This is very useful when using the shell as we keep *state* of our transformation and can peek at our work at the end of every step. On the other hand, once our program works, this multiplication of variables is not as useful and can clutter our program visually.

In PySpark, every transformation returns an object, which is why we need to assign a variable to the result. This means that PySpark doesn't perform modifications *in place*. For instance, the following code block by itself wouldn't do anything because we don't assign the result to a variable or perform an action to display or save our results.

```
results.orderBy("word").count()
```

We can avoid intermediate variables by *chaining* the results of one method to the next. Since each transformation returns a data frame (or a `GroupedData`, when we perform the `groupby()` method, we can directly append the next method without assigning the result to a variable. This means that we can eschew all but one variable assignment. The code in shows the before and after. Note that we also added the `F` prefix to our functions, to respect the import convention we outlined in .

Listing 3.7 Removing intermediate variables by chaining transformation methods

```
# Before
book = spark.read.text("./data/ch02/1342-0.txt")

lines = book.select(split(book.value, " ").alias("line"))

words = lines.select(explode(col("line")).alias("word"))

words_lower = words.select(lower(col("word")).alias("word"))

words_clean = words_lower.select(
    regexp_extract(col("word"), "[a-z']*", 0).alias("word")
)

words_nonull = words_clean.where(col("word") != "")

results = words_nonull.groupby("word").count()

# After

results = (
    spark.read.text("./data/ch02/1342-0.txt")
    .select(F.split(F.col("value"), " ").alias("line"))
    .select(F.explode(F.col("line")).alias("word"))
    .select(F.lower(F.col("word")).alias("word"))
    .select(F.regexp_extract(F.col("word"), "[a-z']*", 0).alias("word"))
    .where(F.col("word") != "")
    .groupby("word")
    .count()
)
```

It’s like night and day: the "after" is much terser and readable, and we’re able to easily follow the list of steps. Visually, we can also see the difference in .

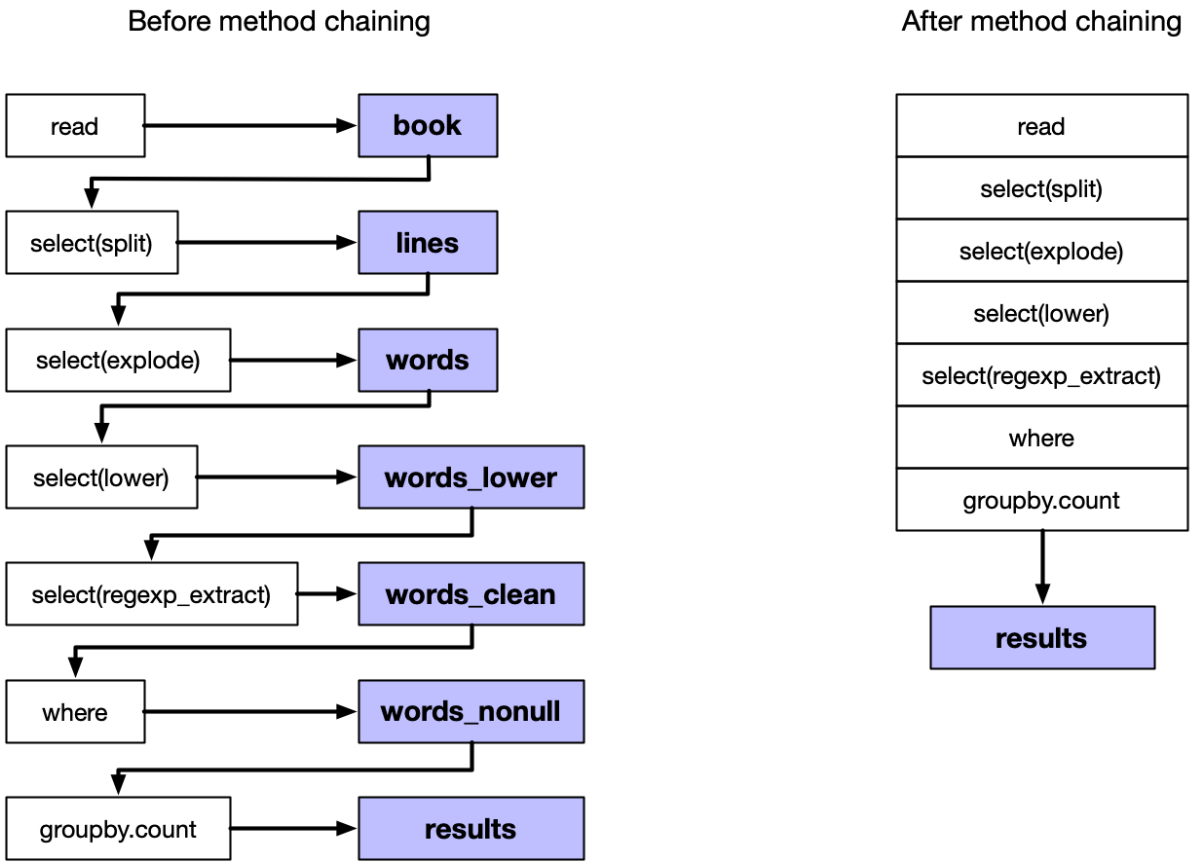


Figure 3.4 Method chaining eliminates the need for intermediate variables

I am not saying that intermediate variables are absolutely evil and to be avoided. They can hinder your code readability, so you have to make sure they serve a purpose. A lot of burgeoning PySpark developers take the habit of always writing on top of the same variable. If you see yourself doing something like , chain your methods like in . You'll get the same result, and prettier code.

Listing 3.8 If you write over the same variable over and over again, consider chaining your methods.

```
# Don't do that

df = spark.read.text("./data/ch02/1342-0.txt")
df = df.select(F.split(F.col("value"), " ").alias("line"))
df = ...
```

SIDEBAR **Make your life easier by using Python's parentheses**

If you look at the "after" code of , you'll notice that I start my right side of the equal sign with an opening parenthesis (`spark = ( […]`). This is a trick I use when I need to chain methods in Python. If you don't wrap your result into a pair of parentheses, you'll need to add a `\` character at the end of each line, which adds visual noise to your program. PySpark code is especially prone to line breaks when you use method chaining.

```
results = spark\
          .read.text('./data/ch02/1342-0.txt')\
          ...
```

As a lazy alternative, I am a big fan of using Black as a Python code formatting tool (https://black.readthedocs.io/). It removes a lot of the guesswork of having your code logically laid-out and consistent. Since we read code more than we write code, readability matters.

Since we are performing two actions on `results` (displaying the top 10 words on the screen and writing the data frame to a csv file), we have to use a variable. If you only have 1 action to perform on your data frame, you can channel your inner code golfer[5] by not using any variable name. Most of the time, I prefer lumping my transformations together and keep the action visually separate, like we are doing now.

Our program is looking much more polished now. The last step will be to add the PySpark's plumbing to prepare it for batch mode.

3.5 Your first non-interactive program: using `spark-submit`

When we launched the `pyspark` shell, we saw that the `spark` variable was mapped to our `SparkSession` entry point, already configured for interactive work. When using batch submit, this isn't the case. In this section, I teach how to create your own entry point and submit the code in batch mode. You will then be able to submit this program (and any properly coded PySpark program).

Before entering in the *how?*, let's see what happens if we submit our program as is. In , we can see that PySpark replies immediately with a `NameError`, saying that `spark` isn't defined.

Listing 3.9 Launching a PySpark program without `spark` defined.

```
$ spark-submit word_count.py
[...]
Traceback (most recent call last):
  File "/Users/jonathan_rioux/Dropbox/PySparkBook/code/Ch02/word_count.py", line 3, in <module>
    results = (spark
NameError: name 'spark' is not defined
[...]
```

Unlike the `pyspark` command, Spark provides a single launcher for its programs in batch mode, called `spark-submit`. The simplest way to submit a program is to provide the program name as the first parameter. As our programs grows in complexity, I will teach through Part 2 and 3 how to augment the `spark-submit` with other parameters.

3.5.1 Creating your own `SparkSession`

In Chapter 1, we learned that our main point of action is through an entry point which in our case is a `SparkSession` object. This section covers how to create a simple bare-bone entry point so our program can run smoothly.

PySpark provides a builder pattern using the object `SparkSession.builder`. For those familiar with object-oriented programming, a builder pattern provides a set of methods to create a highly configurable object without having multiple constructors. In this chapter, we will only look at the happiest case, but the `SparkSession` builder pattern will become increasingly useful in Part 3 as we look into performance tuning and adding dependencies to our jobs.

In , we start the builder pattern, and then then chain a configuration parameter which defined the application name. This isn't absolutely necessary, but when monitoring your jobs (see Chapter 9), having a unique and well thought-out job name will make it easier to know what's what. We finish the builder pattern with the `.getOrCreate()` method to create our SparkSession.

> **NOTE** You can't have two `SparkSession` objects in your program working at the same time. This is why the `getOrCreate()` method is called like this. If you were to create a new entry point in the `pyspark` shell, you'd get all kinds of funny errors. By using the `getOrCreate()` method, your program will work both in interactive and batch mode.

Listing 3.10 Creating our own simple `SparkSession`

```
from pyspark.sql import SparkSession

spark = (SparkSession.builder
                     .appName("Counting word occurences from a book.")
                     .getOrCreate())
```

3.6 Using `spark-submit` to launch your program in batch mode

We saw at the beginning of how to submit a program using `spark-submit`. Let's try again, in , with our properly configured entry point. The full code is available on the book's repository, under `code/Ch02/word_count_submit.py`.

Listing 3.11 Submitting our job for real this time

```
$ spark-submit ./code/Ch02/word_count_submit.py

# [...]
# +----+-----+
# |word|count|
# +----+-----+
# | the| 4480|
# |  to| 4218|
# |  of| 3711|
# | and| 3504|
# | her| 2199|
# |   a| 1982|
# |  in| 1909|
# | was| 1838|
# |   i| 1749|
# | she| 1668|
# +----+-----+
# only showing top 10 rows
# [...]
```

TIP You get a deluge of "INFO" messages? Don't forget that you have control over this: use `spark.sparkContext.setLogLevel("WARN")` right after your `spark` definition. If your local configuration has `INFO` as a default, you'll still get a slew of messages until it catches this line, but it won't obscure your results.

With this, we're done! Our program successfully *ingests* the book, *transforms* it into a cleaned list of word frequencies and then *exports* it two ways: as a top-10 list on the screen and as a CSV file.

If we look at our process, we applied one transformation interactively at the time, `show()`-ing the process after each one. This will often be your *modus operandi* when working with a new data file. Once you're confident about a block of code, you can remove the intermediate variables. PySpark gives you out of the box a productive environment to explore large data sets interactively and provides an expressive and terse vocabulary to manipulate data. It's also easy to go from interactive development to batch deployment: you just have to define your `SparkSession` and you're good to go.

3.7 What didn't happen in this Chapter

Chapter 2 and 3 were pretty dense with information. We learned how to read text data, process it to answer and question, display the results on the screen and write them to a CSV file. On the other hand, there are many elements we left out on purpose. Let's have a quick look at what we *didn't* do in this Chapter.

With the exception of coalescing the data frame in order to write it into a single file, we didn't care much for the distributing of the data. We saw in Chapter 1 that PySpark distributes data

across multiple workers nodes, but our code didn't pay much attention to this. Not having to constantly think about partitions, data locality and fault tolerance made our data discovery process much faster.

We didn't spend much time configuring PySpark. Beside providing a name for our application, no additional configuration was inputted in our `SparkSession`. It's not to say we'll never touch this, but we can start with a bare-bone configuration and tweak as we go. Chapter 6 will expand into the subject.

Finally, we didn't care much about the order of operations. We made a point to describe our transformations as logically as they appear to us, and we're letting Spark's optimize this into efficient processing steps. We could potentially re-order some and get the same output, but our program reads well, is easy to reason about and works well.

This echoes the statement I made in Chapter 1: PySpark is remarkable by not only what it provides, but also what it can abstract over. You can write your code as a sequence of transformations that will get you to your destination most of the time. For those cases where you want more finely-tuned performance or more control about the physical layout of your data, we'll see in Part 3 that PySpark won't hold you back.

3.8 Scaling up our word frequency program

That example wasn't big data. I'll be the first to say it.

Teaching big data processing has a catch 22. While I really want to show the power of PySpark to work with massive data sets, I don't want you to purchase a cluster or rack up a massive cloud bill. It's easier to show the ropes using a smaller set of data, knowing that we can scale using the same code.

Let's take our word counting example: how can we scale this to a larger corpus of text? Let's download more files from Project Gutenberg and place them in the same directory.

```
$ ls -l data/Ch02
[...]
-rw-r--r--@ 1 jonathan_rioux  247087069   173595 Aug  4 15:03 11-0.txt
-rw-r--r--@ 1 jonathan_rioux  247087069   724726 Jul 30 17:30 1342-0.txt
-rw-r--r--@ 1 jonathan_rioux  247087069   607788 Aug  4 15:03 1661-0.txt
-rw-r--r--@ 1 jonathan_rioux  247087069  1276201 Aug  4 15:03 2701-0.txt
-rw-r--r--@ 1 jonathan_rioux  247087069  1076254 Aug  4 15:03 30254-0.txt
-rw-r--r--@ 1 jonathan_rioux  247087069   450783 Aug  4 15:03 84-0.txt
```

While this is not enough to claim "We're doing Big Data(tm)", it'll be enough to explain the general concept. If you really want to scale, you can use Appendix C to provision a powerful cluster on the cloud, download more books or other text files and run the same program for a few dollars.

We will modify our `word_count_submit.py` in a very subtle way. Where we `.read.text()`, we'll just change the path to account for all files in the directory. shows the before and after: we are just changing the `1342-0.txt` to a `*.txt`, which is called a *glob pattern*. This will select all the `.txt` files in the directory.

Listing 3.12 Scaling our word count program

```
# Before
results = (spark
           .read.text('./data/ch02/1342-0.txt')

# After
results = (spark
           .read.text('./data/ch02/*.txt')
```

NOTE You can also just pass the name of the directory if you want PySpark to ingest all the files within the directory.

The results of running the program over all the files in the directory are available in .

Listing 3.13 Results of scaling our program to multiple files

```
$ spark-submit ./code/Ch02/word_count_submit.py

+----+-----+
|word|count|
+----+-----+
| the|38895|
| and|23919|
|  of|21199|
|  to|20526|
|   a|14464|
|   i|13973|
|  in|12777|
|that| 9623|
|  it| 9099|
| was| 8920|
+----+-----+
only showing top 10 rows
```

With this, you can confidently say that you are able to scale a simple data analysis program, using PySpark. You can use the general formula we've outlined here and modify some of the parameters and methods to fit your use-case. Chapter 3 will dig a little deeper into some interesting and common data transformations, building on what we've learned here.

3.9 Summary

- You can group records using the `groupby` method, passing the column names you want to group against as a parameter. This returns a `GroupedData` object that waits for an aggregation method to return the results of a computation over the groups, such as the `count()` of records.
- PySpark's repertoire of functions that operate on columns are located in `pyspark.sql.functions`. The unofficial but well respected convention is to qualify this import in your program using the `F` keyword.
- When writing a data frame to a file, PySpark will create a directory and put one file per partition. If you want to write a single file, use the `coaslesce(1)` method.
- In order to prepare your program to work in batch mode via `spark-submit`, you need to create a `SparkSession`. PySpark provides a builder pattern in the `pyspark.sql` module.
- If your program needs to scale across multiple files within the same directory, you can use a glob pattern to select many files at once. PySpark will collect them in a single data frame.

3.10 Exercises

For this exercise, you'll need the `word_count_submit.py` program we worked on this Chapter. You can pick it from the book's code repository (`Code/Ch03/word_count_submit.py`)

3.10.1 Exercise 3.1

a) Modifying the `word_count_submit.py` program, return the number of distinct words in Jane Austen's *Pride and Prejudice*. (Hint, `results` contains 1 record for each unique word…)

b) (Challenge) Wrap your program in a function that takes a file name as a parameter. It should return the number of distinct words.

3.10.2 Exercise 3.2

Taking `word_count_submit.py`, modify the script to return a sample of 20 words that appear only once in Jane Austen's *Pride and Prejudice*.

3.10.3 Exercise 3.3

a) Using the `substr` function (refer to PySpark's API or the `pyspark` shell help if needed), return the top 5 most popular first letters (keep only the first letter of each word).

b) Compute the number of words starting with a consonant or a vowel. (Hint: the `isin()` function might be useful)

3.10.4 Exercise 3.4

Let's say you want to get both the `count()` and `sum()` of a `GroupedData` object. Why doesn't this code work? Map the inputs and outputs of each method.

```
my_data_frame.groupby("my_column").count().sum()
```

Multiple aggregate function application will be covered in Chapter 4.

4 Analyzing tabular data with pyspark.sql

This chapter covers

- Reading delimited data into a PySpark data frame
- Understanding how PySpark represents tabular data in a data frame
- Ingesting and exploring tabular or relational data
- Selecting, manipulating, renaming and deleting columns in a data frame
- Summarizing data frames for quick exploration

So far, in chapters 2 and 3, we've dealt with textual data, which is *unstructured*. Through a chain of transformations, we extracted some information to get the most common words in the text. This chapter will go a little deeper into data manipulation using *structured* data, which is data that follow a set format. More specifically, we will work with *tabular* data, which follows the classical rows and columns layout. Just like the two previous chapters, we'll take a data set and answer a simple question by exploring and processing the data.

We'll use some public Canadian television schedule data to identify and measure the proportion of commercials over the total programming. The data used is typical of what you see from mainstream relational databases. This chapter and the next builds heavily on chapter 2 and 3, and add additional methods and information to use the data frame as a tabular data container. We perform some data exploration, assembly, and cleaning, using the `pyspark.sql` data manipulation module, and we finish by answering some questions hidden in our data set. The exercises will at the end of the chapter give you the opportunity to craft your own data manipulation code.

The initialization part of our Spark program (relevant imports and `SparkSession` creation) is provided in .

Listing 4.1 Relevant imports and scaffolding for this chapter

```
import os
import numpy as np

from pyspark.sql import SparkSession
import pyspark.sql.functions as F

spark = SparkSession.builder.getOrCreate()
```

4.1 What is tabular data?

Tabular data is data that we can typically represent in a 2-dimensional table. You have rows and columns containing a single (or *simple*) value. A good example would be your grocery list: you may have one column for the item you wish to purchase, one for the quantity, and one for the expected price. provides an example of a small grocery list. We have the three columns mentioned, as well as four rows, each representing an entry in our grocery list.

Item	Quantity	Price
Banana	2	1.74
Apple	4	2.04
Carrot	1	1.09
Cake	1	10.99

Figure 4.1 My grocery list represented as tabular data. Each row represents an item, and each column represents an attribute.

The easiest analogy we can make for tabular data is the spreadsheet format: the interface provides you with a large number of rows and columns where you can input and perform computation on data. SQL databases, even if they use a different vocabulary, can also be thought of as tables made up of rows and columns. Tabular data is an extremely common data format, and because it's so popular and easy to reason about, it makes for a perfect first dive into PySpark's data manipulation API.

PySpark's data frame structure maps very naturally to tabular data. In chapter 2, I explain that PySpark operates either on the whole data frame structure (via methods such as `select()` and `groupby()`) of on `Column` objects (for instance when using a function like `split()`). The data frame is *column-major*, so its API focuses on manipulating the columns to transform the data. Because of this, we can simplify how we reason about data transformations by thinking about what operations to do and which columns will be impacted.

NOTE The resilient distributed dataset, briefly introduced in chapter 1, is a good example of a structure that is *row-major*. Instead of thinking about columns, you are thinking about items with attributes in which you apply functions. It's an alternative way of thinking about your data, and chapter 8 contains a lot more information about where it can be useful.

4.1.1 How does PySpark represent tabular data?

In chapter 2, our data frame always contained a single column, up until the very end where we counted the occurrence of each word. In other words, we took *unstructured* data (a body of text), performed some transformations, and created a two-column table containing the information we wanted. Tabular data is in a way an extension of this, where we have more than one column to work with.

Let's take my very healthy grocery list as an example, and load it into PySpark. To make things simple, we'll encode our grocery list into a list of lists. PySpark has multiple ways to import tabular data, but the two most popular are the list of lists and the pandas data frame. I cover briefly how to work with Pandas in chapter 8 and appendix D contains more information about how to use pandas. Considering the size of our grocery list, it would be a little overkill to import a library just for loading 4 records, so I kept it in a list of lists.

Listing 4.2 Creating a data frame out of our grocery list

```
my_grocery_list = [
    ["Banana", 2, 1.74],
    ["Apple", 4, 2.04],
    ["Carrot", 1, 1.09],
    ["Cake", 1, 10.99],
]    ❶

df_grocery_list = spark.createDataFrame(my_grocery_list, ["Item", "Quantity", "Price"])

df_grocery_list.printSchema()
# root
#  |-- Item: string (nullable = true)     ❷
#  |-- Quantity: long (nullable = true)   ❷
#  |-- Price: double (nullable = true)    ❷
```

❶ My grocery list is encoded in a list of lists.

❷ PySpark automatically inferred the type of each field from the information Python had about each value.

We can easily create a data frame from data in our program with the `spark.createDataFrame` function, as shows. Our first parameter is the data itself. You can either provide a list of items (here a list of lists), a pandas data frame or a Resilient Distributed Dataset, which I cover in chapter 9. The second parameter is the *schema* of the data frame. chapter 6 covers the automatic and manual schema definitions in greater depth. In the meantime, passing a list of column names

will make PySpark happy, while it infers the types (`string`, `long`, and `double`, respectively) of our columns. Visually, the data frame will look like , although much more simplified. The master node knows about the structure of the data frame, but the actual data is represented on the worker nodes. Each column maps to data stored somewhere on our cluster, managed by PySpark. We operate on the abstract structure and let the master delegate the work efficiently.

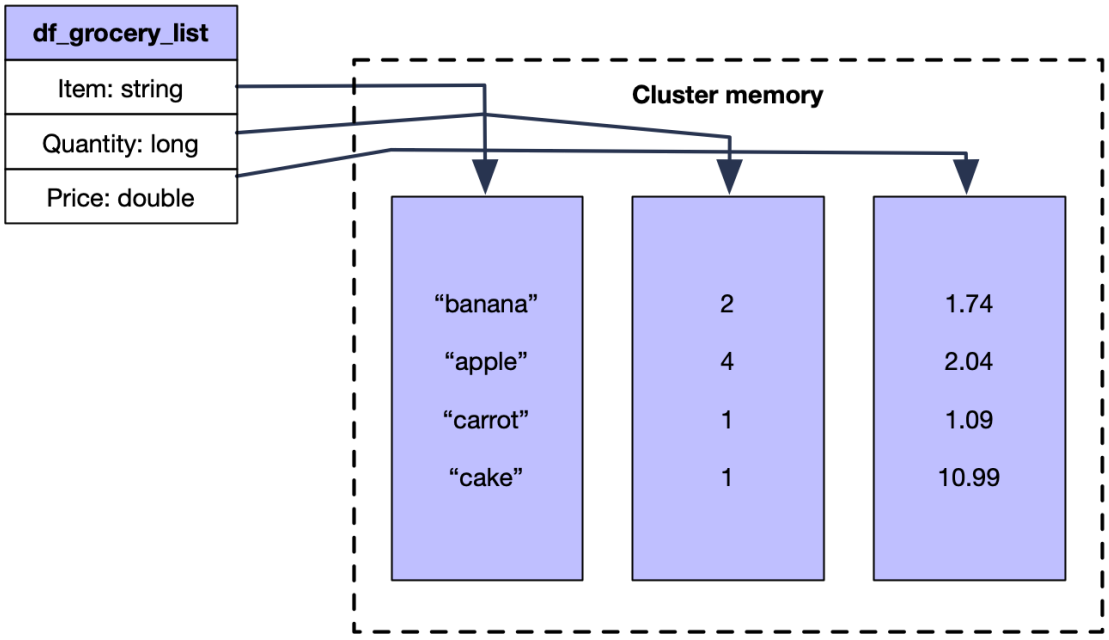


Figure 4.2 Each column of our data frame maps to some place on our worker nodes.

PySpark gladly represented our tabular data using our column definitions. This means that all the functions we learned so far apply to our tabular data. By having one flexible structure for many data representations— we’ve done text and tabular so far— PySpark makes it easy to move from one domain to another. It removes the need to learn yet another set of functions and a whole new abstraction for our data.

4.2 PySpark for analyzing and processing tabular data

My grocery list was fun, but the potential for analysis work is pretty limited. We’ll get our hands on a larger data set, explore it, and ask a few introductory questions that we might find interesting. This process is called *exploratory data analysis* (or EDA) and is usually the first step data analysts and scientists undertake when placed in front of new data. Our goal is to get familiar with the data discovery functions and methods as well as performing some basic data assembly. Being familiar with those steps will remove the awkwardness of working with data you won’t see transforming before your eyes. Until we can process visually millions of records per second, this chapter will show you a blueprint you can re-use when facing new data frames.

SIDEBAR **Graphical exploratory data analysis?**

A lot of the EDA work you'll see in the wild incorporates charts and/or tables. Does it mean that PySpark has the option to do the same?

We saw in chapter 2 how to pretty print a data frame so we can view the content at a glance. This still applies for summarizing information and displaying it on the screen. If you want to export the table in an easy to process format (to incorporate in a report, for instance), you can use `spark.write.csv`, making sure you coalesce the data frame in a single file. (See chapter 3 for a refresher on `coalesce()`.) By its very nature, table summaries won't be very huge so you won't risk running out of memory.

PySpark doesn't provide any charting capabilities and doesn't play with other charting libraries (like matplotlib, seaborn, altair, or plot.ly). Taking a step back, it makes a lot of sense: PySpark distributes your data over many computers. It doesn't make much sense to distribute a chart creation. The usual solution will be to transform your data using PySpark, use the `toPandas()` method to transform your PySpark data frame into a pandas data frame, and then use your favourite charting library. When using charts, I provide the code I used to generate them, but instead of explaining the process each time, I provide explanations— as well as a primer in using pandas with PySpark— in Appendix D.

For this exercise, we'll use some open data from the Government of Canada, more specifically the CRTC (Canadian Radio-television and Telecommunications Commission). Every broadcaster is mandated to provide a complete log of the programs, commercials and all, showcased to the Canadian public. This gives us a lot of potential questions to answer, but we'll select one specific one: **what are the channels with the most and least proportion of commercials?**

You can download the file on the Canada Open Data portal (https://open.canada.ca/data/en/dataset/800106c1-0b08-401e-8be2-ac45d62e662e), selecting the `BroadcastLogs_2018_Q3_M8` file. The file is a whopping 994MB to download, which might be a little too large for some people. The book's repository contains a sample of the data under the `data/Ch04` directory, which you can use in lieu of the original file. You'll also need to download the "Data Dictionary" in DOC form, as well as the "Reference Tables" zip file, unzipping them into a "ReferenceTables" directory in `data/Ch04`. Once again, the examples are assuming that the data is downloaded under `data/Ch04` and that PySpark is launched from `src/Ch04/`.

4.3 Reading delimited data in PySpark

This section is dedicated to ingesting delimited data in a PySpark data frame, so we can start manipulating it. I will cover how to use a specialized reader object for delimited data, the most common parameters to set, and how to identify frequent patterns when looking at tabular data.

Delimited data is a very common, popular, and tricky way of sharing data. In a nutshell, the data is sitting verbatim in a file, separated by two types of delimiters. A visual sample of delimited data is depicted in .

1. The first one is a *row delimiter*. The row delimiter splits the file in logical records. There is one and only one record between delimiters.
2. The second one is a *field delimiter*. Each record is made up of an identical number of fields and the field delimiter tells where one field starts and ends.

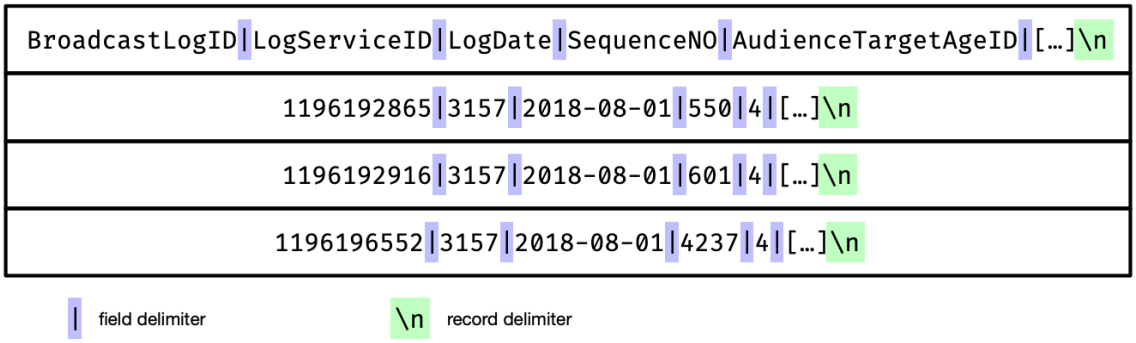


Figure 4.3 A sample of our data, highlighting the field delimiter (|) and row delimiter (\n)

The newline character (`\n` when depicted explicitly) is the de-facto record delimiter. It naturally breaks down the file into visual lines, where one record starts at the beginning of the line and ends, well, at the end. The comma character `,` is the most frequent field delimiter. It's so prevalent that most people call delimited data files "CSV", which stands for "Comma Separated Values".

CSV files are easy to produce and have a loose set of rules to follow to be considered usable. Because of this, PySpark provides a whopping 25 optional parameters when ingesting a CSV. Compare this to the two for reading text data. In , I use three configuration parameters. This is enough to parse our data into a data frame.

Listing 4.3 Reading our broadcasting information

```
DIRECTORY = "../../data/Ch04"
logs = spark.read.csv(
    os.path.join(DIRECTORY, "BroadcastLogs_2018_Q3_M8.CSV"),    ❶
    sep="|",    ❷
    header=True,    ❸
    inferSchema=True,    ❹
)
```

❶ We specify the file path where our data resides first

❷ Our file uses a vertical bar, so we pass `|` as a parameter to `sep`

❸ `header` takes a boolean. When `true`, the first row of your file is parsed as the column names.

❹ `inferSchema` takes a boolean as well. When `true`, it'll pre-parse the data to infer the type of the column.

The next section expands on the most important parameters when reading CSV data and provides more detailed explanations behind the code in .

4.3.1 Customizing the `SparkReader` object to read CSV data files

This section focuses on how we can specialize the `SparkReader` object to read delimited data and what are the most popular configuration parameters to accommodate the various declinations of CSV.

Reading delimited data can be a dicey business. Because of how flexible and human editable the format is, a CSV reader needs to provide many options to cover the many use-cases possible. There is also a risk that the file is malformed, in which case you will need to treat it as text and gingerly infer the fields manually. I will stay on the happy path and cover the most popular scenario: a single file, properly delimited.

THE PATH TO THE FILE YOU WANT TO READ AS THE ONLY MANDATORY PARAMETER

Just like when reading text, the only truly mandatory parameter is the `path`, which contains the file or files path. As we saw in chapter 2, you can use a glob pattern to read multiple files inside a given directory, as long as they have the same structure. You can also explicitly pass a list of file paths if you want specific files to be read.

PASSING AN EXPLICIT FIELD DELIMITER WITH THE `SEP` PARAMETER

The most common variation you'll encounter when ingesting and producing CSV file is selecting the right delimiter. The comma is the most popular, but it suffers from being a popular character in text, which means you need a way to differentiate which commas are part of the text and which oner are delimiters. Our file use the vertical bar character, an apt choice: it's easily reachable on the keyboard yet infrequent in text.

> **NOTE** In French, we use the comma for separating numbers between their integral part and their decimal one (e.g. `1.02` `1,02`). This is pretty awful when in a CSV file, so most French CSV will use the semicolon (`;`) as a field delimiter. This is one more example of why you need to be vigilant when using CSV.

When reading CSV data, PySpark will default to using the comma character as a field delimiter. You can set the optional parameter `sep` (for separator) to the single character you want to use as a field delimiter.

QUOTING TEXT TO AVOID MISTAKING A CHARACTER FOR A DELIMITER

When working with CSV that use the comma as a delimiter, it's common practice to *quote* the text fields to make sure any comma in the text is not mistaken as a field separator. The CSV reader object provides an optional `quote` parameter that defaults to the double-quote character `"`. Since I am not passing an explicit value to `quote`, we are keeping the default value. This way, we can have a field with the value `"Three | Trois"`, whereas we would consider this to be two fields without the quotation character.

If we don't want to use any character as a quote, we need to pass explicitly the empty string to `quote`.

USING THE FIRST ROW AS THE COLUMN NAMES

The `header` optional parameter takes a Boolean flag. If set to true, it'll use the first row of your file (or files, if you're ingesting many) and use it to set your column names.

You also can pass a list of strings as the `schema` optional parameter if you wish to explicitly name your columns. If you don't fill any of those two, your data frame will have `_c*` for column names, where the star is replaced with increasing integers (`_c0`, `_c1`, …).

INFERRING WHAT A COLUMN TYPE WHILE READING THE DATA

PySpark has a schema discovering capacity. You turn it on by setting `inferSchema` to `True` (by default, this is turned off). This optional parameter forces PySpark to go over the ingested data: one time to set the types of each column, one time to ingest the data. This makes the ingestion quite a bit longer but avoids us to write the schema by hand (I go down to this level of detail in chapter 6). Let the machine do the work!

We are lucky enough that the Government of Canada is a good steward of data, and provides us with clean, properly formatted files. In the wild, malformed CSV files are legion and you will run into some errors when trying to ingest some of them. Furthermore, if your data is large, you often won't get the chance to inspect each row one by one to fix mistakes. Chapter 9 covers some strategies to ease the pain and also shows you some ways to share your data with the schema included.

Our data frame schema, displayed on , is coherent with the documentation we've downloaded. The column names are properly displayed and the types make sense. That's plenty enough to get started with some exploration.

Listing 4.4 The schema of our `logs` data frame

```
logs.printSchema()
# root
#  |-- BroadcastLogID: integer (nullable = true)
#  |-- LogServiceID: integer (nullable = true)
#  |-- LogDate: timestamp (nullable = true)
#  |-- SequenceNO: integer (nullable = true)
#  |-- AudienceTargetAgeID: integer (nullable = true)
#  |-- AudienceTargetEthnicID: integer (nullable = true)
#  |-- CategoryID: integer (nullable = true)
#  |-- ClosedCaptionID: integer (nullable = true)
#  |-- CountryOfOriginID: integer (nullable = true)
#  |-- DubDramaCreditID: integer (nullable = true)
#  |-- EthnicProgramID: integer (nullable = true)
#  |-- ProductionSourceID: integer (nullable = true)
#  |-- ProgramClassID: integer (nullable = true)
#  |-- FilmClassificationID: integer (nullable = true)
#  |-- ExhibitionID: integer (nullable = true)
#  |-- Duration: string (nullable = true)
#  |-- EndTime: string (nullable = true)
#  |-- LogEntryDate: timestamp (nullable = true)
#  |-- ProductionNO: string (nullable = true)
#  |-- ProgramTitle: string (nullable = true)
#  |-- StartTime: string (nullable = true)
#  |-- Subtitle: string (nullable = true)
#  |-- NetworkAffiliationID: integer (nullable = true)
#  |-- SpecialAttentionID: integer (nullable = true)
#  |-- BroadcastOriginPointID: integer (nullable = true)
#  |-- CompositionID: integer (nullable = true)
#  |-- Producer1: string (nullable = true)
#  |-- Producer2: string (nullable = true)
#  |-- Language1: integer (nullable = true)
#  |-- Language2: integer (nullable = true)
```

SIDEBAR **Exercise 4.1**

Lets take the following file, called `sample.csv`.

```
Item,Quantity,Price
$Banana, organic$,1,0.99
Pear,7,1.24
$Cake, chocolate$,1,14.50
```

Complete the following code to ingest the file successfully.

```
sample = spark.read.csv([...],
                        sep=[...],
                        header=[...],
                        quote=[...],
                        inferSchema=[...]
)
```

SIDEBAR **Exercise 4.2**

Re-read the data in a `logs_raw` data frame, taking inspiration from the code in , this time without passing any optional parameters. Print the first 5 rows of data, as well as the schema. What are the differences in terms of data and schema between `logs` and `logs_raw`?

4.3.2 Exploring the shape of our data universe

When working with tabular data, especially if it comes from a SQL data warehouse, you'll often find that the data set is split between tables. In our case, our logs table contains a majority of fields suffixed by `ID`; those IDs are listed in other tables and we have to *link* them to get the legend of those IDs. This section introduces briefly what a star schema is, why they are so frequently encountered, and how we can represent them visually to work with them.

Our data universe (the set of tables we are working with) follows a very common pattern in relational databases: a center table containing a bunch of IDs (or *keys*) and some ancillary tables around containing a legend between each key and its value. This is called a *star schema* since it visually looks like a star. Star schemas are common in the relational database world because of *normalization*, a process used to avoid duplicating pieces of data and improve data integrity. Data normalization is illustrated in , where our centre table `logs` contain IDs that maps to the auxiliary tables around called *link tables*. In the case of the `CD_Category` link table, it contains many fields (such as `Category_CD` and `English_description`) that are made available to `logs` when you link the two tables with the `Category_ID` key.

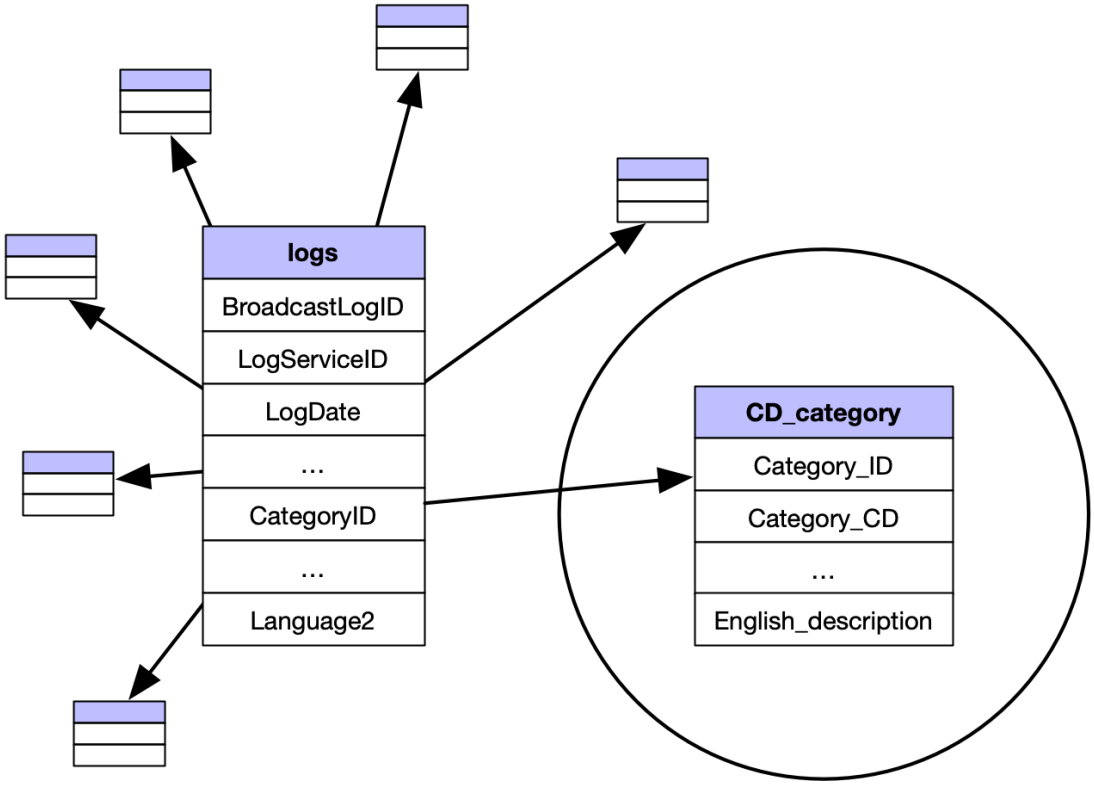


Figure 4.4 The logs table "ID" columns map to other tables, like the CD_category table which links the Category_ID field.

In Spark's universe, we often prefer working with a single table instead of linking a multitude of tables to get the data. We call them *denormalized* tables, or colloquially *fat* tables. We will start by assessing the data available to us directly in the `logs` table before plumping our table, a topic I cover in chapter 5. By looking at the `logs` table, its content, and the data documentation, we will avoid linking tables that contain data with no real value for our analysis.

SIDEBAR **The right structure for the right work**

Normalization, denormalization, what gives? Isn't this a book about data analysis?

While this book isn't about data architecture, it's important to understand, at least a little bit, how data might be structured so we can work with it. Normalized data has many advantages when you're working with relational information (such as our broadcast tables). Besides being easier to maintain, data normalization reduces the probability of getting anomalies or illogical records in your data.

When dealing with analytics, a single table containing all the data is best. However, having to link the data by hand can be tedious, especially when working with dozens or even hundreds of link tables. Fortunately, data warehouses don't change their structure very often. If you're faced with a complex star schema one day, befriend one of the database managers. There is a very good chance that they'll provide you with the information to denormalize the tables, most often in SQL, and chapter 7 will show you how you can adapt the code into PySpark with a minimum of efforts.

4.4 The basics of data manipulation: diagnosing our centre table

It is common practice to explore and summarize the data when you first get acquainted with it. It's just like a first date with your data: you want a good overview, not agonize on the details.[6] This section shows the most common manipulations done on a data frame in greater detail. I show how you can select, delete, rename, re-order, and create columns so you can customize how a data frame is shown. I also cover summarizing a data frame, so you can have a quick diagnostic overview of the data inside your structure. No flowers required.

4.4.1 Knowing what we want: selecting columns

So far, we've learned that typing our data frame variable into the shell prints the structure of the data frame, not the data, unless you're using eagerly evaluated Spark (referenced in chapter 2). We can also use the `show()` command to display a few records for exploration. I won't show the results, but if you try it, you'll see that the table-esque output is garbled, because we are showing too many columns at once. Time to `select()` our way to sanity.

At its simplest, `select()` can take one or more column objects— or strings representing column names— and will return a data frame containing only the listed columns. This way, we can keep our exploration tidy and check a few columns at the time. An example is displayed in .

Listing 4.5 Selecting 5 rows of the first 3 columns of our data frame

```
logs.select(*logs.columns[:3]).show(5, False)

# +--------------+------------+-------------------+
# |BroadcastLogID|LogServiceID|LogDate            |
# +--------------+------------+-------------------+
# |1196192316    |3157        |2018-08-01 00:00:00|
# |1196192317    |3157        |2018-08-01 00:00:00|
# |1196192318    |3157        |2018-08-01 00:00:00|
# |1196192319    |3157        |2018-08-01 00:00:00|
# |1196192320    |3157        |2018-08-01 00:00:00|
# +--------------+------------+-------------------+
# only showing top 5 rows
```

In chapter 2, you learn that `.show(5, False)` shows 5 rows without truncating their representation so we can show the whole content. The `.select()` statement is where the magic happens.

Each data frame created keeps a list of column names in its `columns` attribute, and we can slice it to pick a subset of columns to see. We're using the implicit string to column conversion PySpark provides to avoid some boilerplate.

`select()`, takes a single parameter, named `*cols`. This star is used in Python for unpacking collections, or in our case to illustrate that the function takes a variable number of parameters that will be collected under the `cols` variable. From a PySpark perspective, the four statements in are interpreted the same. Note how prefixing the list with a star removed the container so each element becomes a parameter of the function. If this looks a little confusing to you, fear not! Appendix D will provide you with a good overview of its usage.

Listing 4.6 Four ways to select colums in PySpark, all equivalent in term of results

```
# Using the string to column conversion
logs.select("BroadCastLogID", "LogServiceID", "LogDate")
logs.select(*["BroadCastLogID", "LogServiceID", "LogDate"])

# Passing the column object explicitly
logs.select(F.col("BroadCastLogID"), F.col("LogServiceID"), F.col("LogDate"))
logs.select(*[F.col("BroadCastLogID"), F.col("LogServiceID"), F.col("LogDate")])
```

When explicitly selecting a few columns, you don't have to wrap them into a list. If you're already working on a list of columns, you can unpack them with a star prefix. This argument unpacking pattern is worth remembering as many other data frame methods taking columns as input are using the same approach.

In the spirit of being clever (or lazy), let's expand our selection code to see every column in groups of three. This will give us a sense of the content. Since `logs.columns` is a Python list, we can use a function on it without any problem. The code in shows one of the ways we can do it.

Listing 4.7 Peeking at the data frame in chunks of 3 columns

```
column_split = np.array_split(np.array(logs.columns), len(logs.columns) // 3)    ❶

print(column_split)

# [array(['BroadcastLogID', 'LogServiceID', 'LogDate'], dtype='<U22'),
#  [...]
#  array(['Producer2', 'Language1', 'Language2'], dtype='<U22')]'

for x in column_split:
    logs.select(*x).show(5, False)

# +--------------+------------+-------------------+
# |BroadcastLogID|LogServiceID|LogDate            |
# +--------------+------------+-------------------+
# |1196192316    |3157        |2018-08-01 00:00:00|
# |1196192317    |3157        |2018-08-01 00:00:00|
# |1196192318    |3157        |2018-08-01 00:00:00|
# |1196192319    |3157        |2018-08-01 00:00:00|
# |1196192320    |3157        |2018-08-01 00:00:00|
# +--------------+------------+-------------------+
# only showing top 5 rows
# ... and more tables of 3 columns
```

❶ The `split_array()` function comes from the `numpy` package, imported as `np` at the beginning of this chapter.

Let's take each line one at a time.

We start by splitting the `logs.columns` list into approximate groups of 3. To do so, we rely on a function from the `numpy` package called `array_split()`. The function takes an array and a number of desired sub-arrays `N` and returns a list of `N` sub-arrays. We wrap our list of columns `logs.columns` into an array via the `np.array` function and pass this as a first parameter. For the number of sub-arrays, we divide the number of columns by 3, using an integer division `//`.

TIP To be perfectly honest, the call to `np.array` can be eschewed since `np.array_split()` can work on lists. I am still using it since if you are using a static type checker, such as mypy, you'll get a type error. Chapter 8 has a basic introduction to type checking in Python and Appendix D provides a little more guidance on how to type check your code.

The last part of iterates over the list of sub-arrays, using `select()` so select the columns present inside each sub-array and `show()` to display them on the screen.

This example shows how easy it is to blend Python code with PySpark. On top of providing a trove of functions, the data frame API also exposes information, such as column names, into convenient Python structures. I won't avoid using functionality from libraries when it makes sense, but like in , I'll do my best to explain what it does and why we're using it. Chapter 8 goes beyond on the subject of how you can further combine pure Python code in PySpark.

In this section, we used the `select()` method to rapidly peek at a sample of our data. Because of the width of our data frame, we split our columns into manageable sets of three to keep the output tidy on the screen. I use this pattern frequently to have a high-level view of what my data frame contains. Next, we do the opposite: specifying what we do not want to keep.

4.4.2 Keeping what we need: deleting columns

The other side of selecting columns is choosing what not to select. We could do the full trip with `select()`, carefully crafting our list of columns to keep only the one we want. Fortunately, PySpark also provides a shorter trip: just drop what you don't want.

In our current data frame, let's get rid of two columns in the spirit of tidying up. Hopefully, it will bring us joy.

- `BroadCastLogID` is the primary key of the table and will serve us no use in answering our questions.
- `SequenceNo` is a sequence number and won't be useful either.

More will come off later when we start looking at the link tables. The code in does the trick very simply.

Listing 4.8 Getting rid of columns using the `drop()` method

```
logs = logs.drop("BroadcastLogID", "SequenceNO")

# Testing if we effectively got rid of the columns

print("BroadcastLogID" in logs.columns)  # => False
print("SequenceNo" in logs.columns)  # => False
```

Just like `select()`, `drop()` takes a `*cols` and returns a data frame, this time excluding the columns passed as parameters. Just like every other method in PySpark, `drop()` returns a new data frame, so we overwrite our `logs` variable by assigning the result of our code.

WARNING Unlike `select()`, where selecting a column that doesn't exist will return a runtime error, dropping a non-existent column is a no-op. PySpark will just ignore the columns it doesn't find. Careful with the spelling of your column names!

Depending on how many columns you want to preserve, select might be a neater way to keep just what you want. We can see `drop()` and `select()` as being two sides of the same coin: one drops what you specify, the other one keeps what you specify. We could reproduce with a `select()` method, and does just that.

Listing 4.9 Getting rid of columns, select-style

```
logs = logs.select(
    *[x for x in logs.columns if x not in ["BroadcastLogID", "SequenceNO"]]
)
```

SIDEBAR **Advanced topic: An unfortunate inconsistency**

In theory, you can also `select()` columns with a list without unpacking it. This code will work as expected.

```
logs = logs.select(
    [x for x in logs.columns if x not in ["BroadcastLogID", "SequenceNO"]]
)
```

This is not the case for `drop()`, where you need to explicitly unpack.

```
logs.drop(logs.columns[:])
# TypeError: col should be a string or a Column

logs.drop(*logs.columns[:])
# DataFrame[]
```

I'd rather unpack explicitly and avoid the cognitive load of remembering when it's mandatory and when it's optional.

You now know the most fundamental operations to perform on a data frame. You can select and drop columns, and with the flexibility of `select()` presented in chapters 2 and 3, you can apply functions on existing columns to transform them. The next section will cover how you can create new columns without having to rely on `select()`, simplifying your code, and improving its resiliency.

SIDEBAR **Exercise 4.3**

Create a new data frame `logs_clean` that contains only the columns that do not end with `ID`.

SIDEBAR **Exercise 4.4**

What is the printed result of this code?

```
sample_frame.columns

# ['item', 'price', 'quantity', 'UPC']

print(sample_frame.drop('item', 'UPC', 'prices').columns)
```

1. `['item' 'UPC']`
2. `['item', 'upc']`
3. `['price', 'quantity']`
4. `['price', 'quantity', 'UPC']`
5. Raises an error

4.4.3 Creating what's not there: new columns with `withColumn()`

Creating new columns is such a basic operation that it seems a little far-fetched to rely on `select()`. It also puts a lot of pressure on code readability: for instance using `drop()` makes it obvious we're removing some columns. It would be nice to have something that signals we're creating a new column. PySpark named this function `withColumn()`.

Before going crazy with column creation, let's take a simple example, build what we need iteratively and then move them to `withColumn()`. Let's take the `Duration` column, containing the length of each program shown.

```
logs.select(F.col("Duration")).show(5)

# +----------------+
# |        Duration|
# +----------------+
# |02:00:00.0000000|
# |00:00:30.0000000|
# |00:00:15.0000000|
# |00:00:15.0000000|
# |00:00:15.0000000|
# +----------------+
# only showing top 5 rows

print(logs.select(F.col("Duration")).dtypes)    ❶

# [('Duration', 'string')]
```

PySpark doesn't have a default type for time without dates or duration, so it kept the column as a string. We verified the exact type via the `dtypes` attribute, which returns both the name and type of a data frame's columns. A string is a safe and reasonable option, but this isn't remarkably useful for our purpose. Thanks to our peeking, we can see that the string is formatted like `HH:MM:SS.mmmmmm`, where

- `HH` is the duration in hours
- `MM` is the duration in minutes
- `SS` is the duration in seconds
- `mmmmmmm` is the duration in milliseconds

I ignore the duration in milliseconds since I don't think it'll make a huge difference. In , we are extracting the three other sub-fields.

Listing 4.10 Extracting the hours, minutes and seconds from the `Duration` column

```
logs.select(
    F.col("Duration"),    ❶
    F.col("Duration").substr(1, 2).cast("int").alias("dur_hours"),    ❷
    F.col("Duration").substr(4, 2).cast("int").alias("dur_minutes"),    ❸
    F.col("Duration").substr(7, 2).cast("int").alias("dur_seconds"),    ❹
).distinct().show(    ❺
    5
)

# +----------------+---------+-----------+-----------+
# |        Duration|dur_hours|dur_minutes|dur_seconds|
# +----------------+---------+-----------+-----------+
# |00:10:06.0000000|        0|         10|          6|
# |00:10:37.0000000|        0|         10|         37|
# |00:04:52.0000000|        0|          4|         52|
# |00:26:41.0000000|        0|         26|         41|
# |00:08:18.0000000|        0|          8|         18|
# +----------------+---------+-----------+-----------+
# only showing top 5 rows
```

❶ The original column, for sanity

❷ The first two characters are the hours

❸ The fourth and fifth characters are the minutes

❹ The seventh and eighth characters are the seconds

❺ To avoid seeing identical rows, I've added a `distinct()` to the results

The `substr()` method takes two parameters. The first gives the position of where the sub-string starts, the first character being `1`, not `0` like in Python. The second gives the length of the sub-string we want to extract in number of characters. Following the application of `substr()`, we then cast the result as `int` (integer) using the `cast` method so we can treat them as integers. Casting is a very common and important operation and is covered in more detail in chapter 6. We finally provided an alias for each column so we know easily which one is which.

I used before `show()` the `distinct()` method, which de-dupes the data frame. This is explained further in chapter 5. I added `distinct()` to avoid seeing identical occurrences that would provide no additional information when displayed.

I think that we're in good shape! Let's merge all those values into a single field: the duration of the program in seconds. PySpark can perform arithmetic with column objects using the same operators as Python, so this will be a breeze! The code in takes the code forming the additional

columns in and use it in the definition of a single column.

Listing 4.11 Creating a duration in second field from the `Duration` column

```
logs.select(
    F.col("Duration"),
    (
        F.col("Duration").substr(1, 2).cast("int") * 60 * 60
        + F.col("Duration").substr(4, 2).cast("int") * 60
        + F.col("Duration").substr(7, 2).cast("int")
    ).alias("Duration_seconds"),
).distinct().show(5)

# +----------------+----------------+
# |        Duration|Duration_seconds|
# +----------------+----------------+
# |00:10:30.0000000|             630|
# |00:25:52.0000000|            1552|
# |00:28:08.0000000|            1688|
# |06:00:00.0000000|           21600|
# |00:32:08.0000000|            1928|
# +----------------+----------------+
# only showing top 5 rows
```

We kept the same definitions, removed the alias, and performed arithmetic directly on the columns. There are 60 seconds in a minute, and 60 * 60 seconds in an hour. PySpark respects operator precedence, so we don't have to clobber our equation with parentheses. Overall, our code is quite easy to follow and we are ready to add our column to our data frame.

Instead of using `select()` on all the columns *plus* our new one, let's use `withColumn()`. Applied to a data frame, it'll return a data frame with the new column appended. takes our field and add it to our `logs` data frame. I also include a sample of the `printSchema()` so you can see the column added at the end.

Listing 4.12 Creating a new column with `withColumn()`

```
logs = logs.withColumn(
    "Duration_seconds",
    (
        F.col("Duration").substr(1, 2).cast("int") * 60 * 60
        + F.col("Duration").substr(4, 2).cast("int") * 60
        + F.col("Duration").substr(7, 2).cast("int")
    ),
)

logs.printSchema()

# root
#  |-- LogServiceID: integer (nullable = true)
#  |-- LogDate: timestamp (nullable = true)
#  |-- AudienceTargetAgeID: integer (nullable = true)
#  |-- AudienceTargetEthnicID: integer (nullable = true)
#  [... more columns]
#  |-- Language2: integer (nullable = true)
#  |-- Duration_seconds: integer (nullable = true)    ❶
```

❶ Our `Duration_seconds` colums has been added at the end of our data frame.

WARNING If you're creating a column `withColumn()` and give it a name that already exists in your data frame, PySpark will happily overwrite the column. This is often very useful to keep the number of columns manageable, but make sure you are seeking this effect!

We can create columns using the same expression with `select()` and with `withColumn()`. Both approaches have their use. `select()` will be useful when you're explicitly working with a few columns. When you need to create new ones without changing the rest of the data frame, prefer `withColumn()`. You'll quickly get the intuition about which one is easiest to use when faced with the choice.

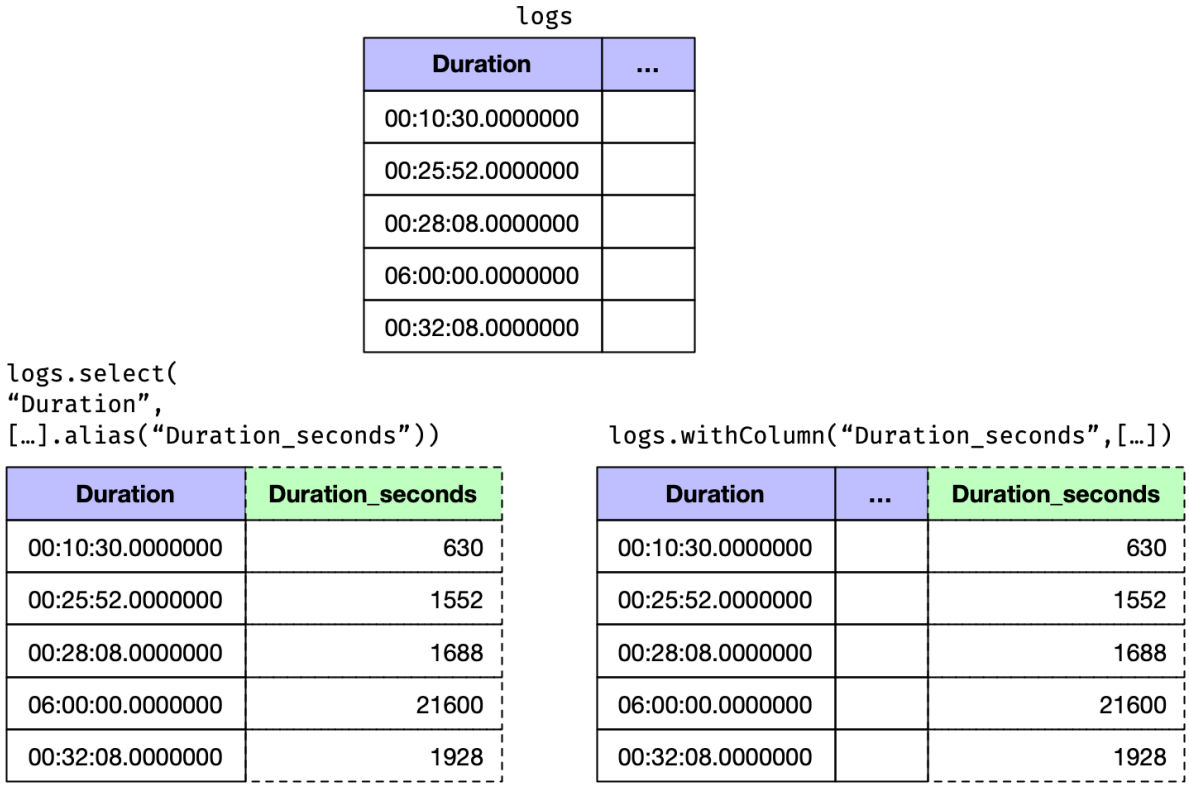


Figure 4.5 select() vs. withColumn(), visually. withColumn() keeps all the pre-existing columns without the need the specify them explicitly.

4.4.4 Tidying our data frame: renaming and re-ordering columns

This section covers how to make the order and name of the columns friendlier for you. It might seem a little vapid, but after a few hours of hammering code on a particularly tough piece of data, you'll be happy to have this in your toolbox.

Renaming columns can be done with `select()` and `alias`, of course. We saw briefly in chapter 2 that PySpark provides you an easier way to do so. Enter `withColumnRenamed()`! In , I use

`withColumnRenamed()` to remove the capital letters of my newly created `duration_seconds` column.

Listing 4.13 Renaming one column at a type, the `withColumnRenamed()` way

```
logs = logs.withColumnRenamed("Duration_seconds", "duration_seconds")

logs.printSchema()

# root
#  |-- LogServiceID: integer (nullable = true)
#  |-- LogDate: timestamp (nullable = true)
#  |-- AudienceTargetAgeID: integer (nullable = true)
#  |-- AudienceTargetEthnicID: integer (nullable = true)
#  [...]
#  |-- Language2: integer (nullable = true)
#  |-- Duration_seconds: integer (nullable = true)
```

I'm a huge fan of having column names without capital letters. I'm a lazy typist, and pressing Shift all the time really adds up folks! I could potentially use `withColumnRenamed()` with a for loop over all the columns to rename all my columns in my data frame. The PySpark developers thought about this and offered a better way to rename all the columns of your data frame in one fell swoop. This relies on a method, `toDF()`, that returns a new data frame with the new columns. Just like `drop()`, `toDF()` takes a `*cols` so we'll need to unpack our column names if they're in a list. The code in shows how you can rename all your columns to lower case in a single line using that method.

Listing 4.14 Batch lower-casing your data frame using the `toDF()` method

```
logs.toDF(*[x.lower() for x in logs.columns]).printSchema()

# root
#  |-- logserviceid: integer (nullable = true)
#  |-- logdate: timestamp (nullable = true)
#  |-- audiencetargetageid: integer (nullable = true)
#  |-- audiencetargetethnicid: integer (nullable = true)
#  |-- categoryid: integer (nullable = true)
#  [...]
#  |-- language2: integer (nullable = true)
#  |-- duration_seconds: integer (nullable = true)
```

If you look at the line of code, you can see that I'm not assigning the resulting data frame. I wanted to showcase the functionality, but since we have ancillary tables with column names that match, I wanted to avoid the trouble of lower-casing every column in every table.

Our final step is *re-ordering* columns. Since re-ordering columns is equivalent to selecting columns in a different order, `select()` is the perfect method for the job. For instance, if we wanted to sort the columns alphabetically, we can use the `sorted` function on the list of our data frame columns, just like in .

Listing 4.15 Selecting our columns in alphabetical order using `select()`

```
logs.select(sorted(logs.columns)).printSchema()

# root
#  |-- AudienceTargetAgeID: integer (nullable = true)
#  |-- AudienceTargetEthnicID: integer (nullable = true)
#  |-- BroadcastOriginPointID: integer (nullable = true)
#  |-- CategoryID: integer (nullable = true)
#  |-- ClosedCaptionID: integer (nullable = true)
#  |-- CompositionID: integer (nullable = true)
#  [...]
#  |-- Subtitle: string (nullable = true)
#  |-- duration_seconds: integer (nullable = true)   ❶
```

❶ Remember that, in most programming languages, capital letters comes before lower-case ones.

4.4.5 Summarizing your data frame: `describe()` and `summary()`

When working with numerical data, looking at a long column of values isn't very useful. We're often more concerned about some key information, which may include count, mean, standard deviation, minimum, and maximum. The `describe()` method does exactly that. By default, it will show those statistics on all numerical and string columns, which will overflow our screen and be unreadable since we have many columns. I display the columns description one-by-one by iterating over the list of columns and showing the output of `describe()` in . Note that `describe()` will (lazily) compute the data frame but won't display it, so we have to `show()` the result.

Listing 4.16 Describing everything in one fell swoop

```
for i in logs.columns:
    logs.describe(i).show()

# +-------+------------------+   ❶
# |summary|      LogServiceID|
# +-------+------------------+
# |  count|           7169318|
# |   mean|3453.8804215407936|
# | stddev|200.44137201584468|
# |    min|              3157|
# |    max|              3925|
# +-------+------------------+
#
# [...]
#
# +-------+   ❷
# |summary|
# +-------+
# |  count|
# |   mean|
# | stddev|
# |    min|
# |    max|
# +-------+

# [... many more little tables]
```

❶ Numerical columns will display the information in a description table like so.

❷ If the type of the column isn't compatible, PySpark displays only the title column.

It will take more time than doing everything in one fell swoop, but the output will be a lot friendlier. Now, because the mean or standard deviation of a string is not a thing, you'll see null values here. Furthermore, some columns won't be displayed (you'll see time tables with only the title column), as `describe()` will only work for numerical and string columns. For a short line to type, you still get a lot!

`describe()` is a fantastic method, but what if you want more? `summary()` at the rescue!

Where `describe()` will take `*cols` as a parameter (one or more columns, the same way as `select()` or `drop()`), `summary()` will take `*statistics` as a parameter. This means that you'll need to select the columns you want to see before passing the `summary()` method. On the other hand, we can customize the statistics we want to see. By default, `summary()` shows everything `describe()` shows, adding the approximate 25%-50% and 75% percentiles. shows how you can replace `describe()` for `summary()` and the result of doing so.

Listing 4.17 Summarizing everything in one fell swoop: default or custom options.

```
for i in logs.columns:
    logs.select(i).summary().show()    ❶

# +-------+------------------+
# |summary|      LogServiceID|
# +-------+------------------+
# |  count|           7169318|
# |   mean|3453.8804215407936|
# | stddev|200.44137201584468|
# |    min|              3157|
# |    25%|              3291|
# |    50%|              3384|
# |    75%|              3628|
# |    max|              3925|
# +-------+------------------+
#
# [... many more slightly larger tables]

for i in logs.columns:
    logs.select(i).summary("min", "10%", "90%", "max").show()    ❷

# +-------+------------+
# |summary|LogServiceID|
# +-------+------------+
# |    min|        3157|
# |    10%|        3237|
# |    90%|        3710|
# |    max|        3925|
# +-------+------------+
#
# [...]
```

❶ By default, we have `count`, `mean`, `stddev`, `min`, `25%`, `50%`, `75%`, `max` as statistics.

❷ We can also pass our own following the same nomenclature convention.

If you want to limit yourself to a subset of those metrics, `summary()` will accept a number of string parameters representing the statistic. You can input `count`, `mean`, `stddev`, `min` or `max` directly. For approximate percentiles, you need to provide them in `XX%` format, such as `25%`.

Both methods will work only on non-null values. For the summary statistics, it's the expected behaviour, but the "count" entry will also count only the non-null values for each column. A good way to see which columns are mostly empty!

WARNING `describe()` and `summary()` are two very useful methods, but they are not meant to be used anywhere else than during development, to quickly peek at data. The PySpark developers don't guarantee the backward compatibility of the output, so if you need one of the outputs for your program, use the corresponding function in `pyspark.sql.functions`. They're all there.

This chapter covered the ingestion and discovery of a tabular data set, one of the most popular data representation formats. We built on the basics of PySpark data manipulation, covered in chapters 2 and 3, and added a new layer by working with columns. The next chapter will be the direct continuation of this one, where we will explore more advanced aspects of the data frame structure.

4.5 Summary

- PySpark uses the `SparkReader` object to read any kind of data directly in a data frame. The specialized `` `CSV `` `SparkReader` is used to ingest comma-separated value (CSV) files. Just like when reading text, the only mandatory parameter is the source location.
- The CSV format is very versatile, so PySpark provides many optional parameters to account for this flexibility. The most important ones are the field delimiter, the record delimiter, and the quotation character. All of those parameters have sensible defaults.
- PySpark can infer the Schema of a CSV file by setting the `inferSchema` optional parameter to `True`. PySpark accomplishes this by reading the data twice: once for setting the appropriate types for each columns, and another time to ingest the data in the inferred format.
- Tabular data is represented into a data frame in a series of Columns, each having a name and a type. Since the data frame is a column-major data structure, the concept of row is less relevant.
- You can use Python code to explore the data efficiently, using the column list as any Python list to expose the elements of the data frame of interest.
- The most common operations on a data frame are the selection, deletion, and creation or columns. In PySpark, the methods used are `select()`, `delete()` and `withColumn()`, respectively.
- `select` can be used for column re-ordering by passing a re-ordered list of columns.
- You can rename columns one by one with the `withColumnRenamed()` method, or all at once by using the `toDF()` method.
- You can display a summary of the columns with the `describe()` or `summary()` method. `describe()` has a fixed set of metrics, while `summary()` will take functions as parameters and apply them to all columns.

5 *The data frame through a new lens: joining and grouping*

In this chapter, you will learn

- Joining two data frames together.
- How to select the right type of join for your use-case.
- Grouping data and understanding the `GroupedData` transitional object.
- Breaking the `GroupedData` with an aggregation method and getting a summarized data frame.
- Filling null values in your data frame

In chapter 4, we looked at how we can transform a data frame using selection, dropping, creation, renaming, re-ordering, and summary of columns. Those operations constitute the foundation working with a data frame in PySpark. In this chapter, I will complete the review of the most common operations you will perform on a data frame: linking or *joining* data frames together, as well as grouping data (and performing operations on the `GroupedData` object). We conclude this chapter by wrapping our exploratory program into a single script we can submit, just like we performed in chapter 3.

We use the same `logs` table that we left in chapter 4. In practical steps, this chapter's code enriches our table with the relevant information contained in the link tables and then get summarized into relevant groups, using what can be considered a graduate version of the `describe()` method I show in chapter 4. If you want to catch up with a minimal amount of fuzz, I provide an `end_of_chapter.py` script in the `src/Ch04` directory.

5.1 From many to one: joining data

When working with data, we're most often working on one structure at a time. Up until now, we've been exploring the many ways we can slice, dice, and modify a data frame to our wildest desires. What happens when we need to link two sources together? This section will introduce joins and how we can apply them when using a star schema setup or another set of tables where values match exactly. This is the easiest use case: joins can get dicey, and chapter 9 drills deeper into the subject of efficient joins.

Joining data frames is a common operation when working with related tables together. If you've used other data processing libraries, you might have seen the same operation being called a *merge* or a *link*. Joins are very common operations, but they are very flexible: in the next section, we set a common vocabulary to avoid confusion and build our understanding on a solid foundation.

5.1.1 What's what in the world of joins

At its most basic, a join operation is a way to take the data from a data frame and add it to another one according to a set of rules. To introduce the moving parts of a join in a practical fashion, I ingest in listing 5.1 a second table to be joined to our `logs` data frame. I use the same parametrization of the `SparkReader.csv` as used for the `logs` table to read our new `log_identifier` table. Once ingested, I filter the data frame so keep only the primary channels, as per the data documentation. With this, we should be good to go.

Listing 5.1 Exploring our first link table: log_identifier

```
DIRECTORY = "../../data/Ch04"
log_identifier = spark.read.csv(
    os.path.join(DIRECTORY, "ReferenceTables/LogIdentifier.csv"),
    sep="|",
    header=True,
    inferSchema=True,
)

log_identifier.printSchema()
# root
#  |-- LogIdentifierID: string (nullable = true)   ❶
#  |-- LogServiceID: integer (nullable = true)   ❷
#  |-- PrimaryFG: integer (nullable = true)   ❸

log_identifier.show(5)
# +--------------+------------+---------+
# |LogIdentifierID|LogServiceID|PrimaryFG|
# +--------------+------------+---------+
# |           13ST|        3157|        1|
# |         2000SM|        3466|        1|
# |           70SM|        3883|        1|
# |           80SM|        3590|        1|
# |           90SM|        3470|        1|
# +--------------+------------+---------+
# only showing top 5 rows

log_identifier = log_identifier.where(F.col("PrimaryFG") == 1)
log_identifier.count()
# 920
```

❶ Channel identifier

❷ Channel key (maps to our centre table!)

❸ Boolean flag: is the channel primary (1) or not (0)? We want only the 1's

We have two data frames, containing each a set of columns. The join operation has three major ingredients:

1. two tables, called a *left* and a *right* table respectively (more on this in a moment);
2. one or more *predicates* which are the series of conditions that determine if records between the two tables are joined;
3. finally, a *method* to indicate how we perform the join when the predicate succeeds and when it fails.

With those three ingredients, you can construct a join between two data frames in PySpark by filling the `[KEYWORDS]` with the desired behaviour. In listing 5.2, I show what a barebone join recipe looks like before we start digging on how to customize those ingredients. The next three sections are dedicated to each one of those parameters.

Listing 5.2 A barebone recipe for a join in PySpark: we need to replace the `[KEYWORDS]` block.

```
[LEFT].join(
    [RIGHT],
    on=[PREDICATES]
    how=[METHOD]
)
```

5.1.2 Knowing our left from our right

A join is performed on two tables at a time. Because of the SQL heritage in the vocabulary of data manipulation, the two tables are being named *left* and *right* tables. Chapter 7 contains a little more explanation about why they are called this way. In PySpark, a neat way to remember which one is which is to say that the left table is *to the left* of the `join()` method where the right one is *to the right* (inside the parentheses). Knowing which one is which is very useful when choosing the join method: unsurprisingly, there is a left and right join type…

Our tables are now identified, so we can update our join blueprint as I do in listing 5.3. We now need to steer our attention to the next parameter, the predicates.

Listing 5.3 A barebone recipe for a join in PySpark, with the left and right tables filled in.

```
logs.join(
    log_identifier,
    on=[PREDICATES]
    how=[METHOD]
)
```

5.1.3 The rules to a successful join: the predicates

The predicates of a PySpark join are rules between columns of the left and right data frames. A join is performed record-wise, where each record on the left data frame is compared (via the predicates) to each record on the right data frame. If the predicates return `True`, the join is a match and is a failure if `False`.

The best way to illustrate a predicate is to create a simple example and explore the results. For our two data frames, we will build the predicate `logs["LogServiceID"] == log_identifier["LogServiceID"]`. In plain English, we can translate this by the following.

Match the records from the `logs` data frame to the records from the `log_identifier` data frame when the value of their `LogServiceID` column is equal.

I've taken a small sample of the data in both data frames and illustrated the result of applying the predicate in figure 5.1. There are two important points to highlight:

1. If one record in the left table resolves the predicate with more than one record in the right

table (or vice versa), this record will be duplicated in the joined table.

2. If one record in the left or in the right table does not resolve the predicate with any record in the other table, it will not be present in the resulting table, *unless the join method specifies a protocol for failed predicates*.

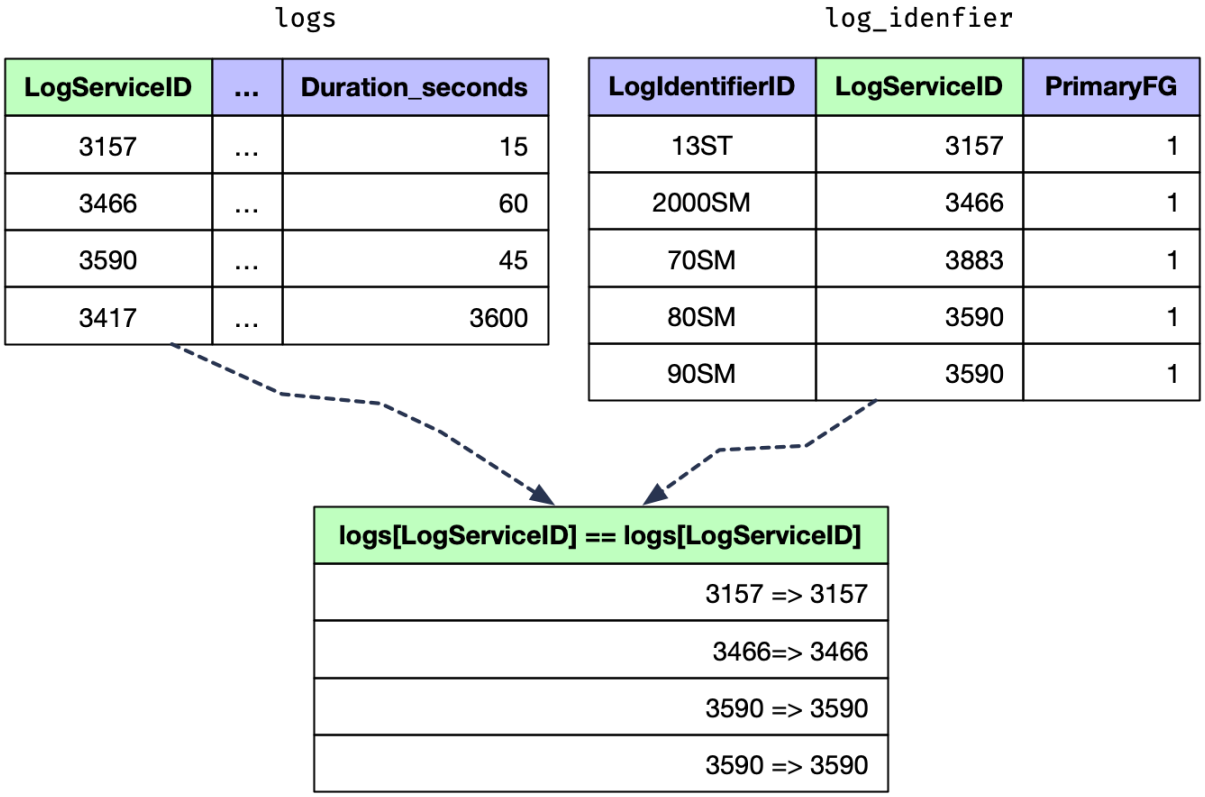


Figure 5.1 Our predicate is applied to a sample of our two tables. 3590 on the left table resolves the predicate twice while 3417 on the left/3883 on the right do not solve it.

In our example, the `3590` record on the left is equal to the two corresponding records on the right, and we see two solved predicates with this number in our result set. On the other hand, the `3417` record does not match anything on the right and therefore is not present in the result set. The same thing happens with the `3883` record on the right table.

You are not limited to a single test in your predicate. You can use multiple conditions by separating them with boolean operators such as `|` (or) or `&` (and). You can also use a different test than equality. Here are two examples and their plain English translation.

1. `(logs["LogServiceID"] == log_identifier["LogServiceID"]) & (logs["left_col"] < log_identifier["right_col"])` : Will only match the records that have the same `LogServiceID` on both side **and** where the value of the `left_col` in the `logs` table is smaller than the value of the `right_col` in the `log_identifier` table.
2. `(logs["LogServiceID"] == log_identifier["LogServiceID"]) | (logs["left_col"] > log_identifier["right_col"])` : Will only match the

records that have the same `LogServiceID` on both side **or** where the value of the `left_col` in the `logs` table is greater than the value of the `right_col` in the `log_identifier` table.

You can build the operations as complicated as you want. I really recommend wrapping each condition in parentheses to avoid worrying about the operator precedence and facilitate the reading.

Before adding our predicate to our join in progress, I want to note that PySpark provides a few predicate shortcuts to reduce the complexity of the code.

If **you have multiple and predicates** (such as `(left["col1"] == right["colA"]) & (left["col2"] > right["colB"]) & (left["col3"] != right["colC"])`), you can put them into a list such as `[left["col1"] == right["colA"], left["col2"] > right["colB"], left["col3"] != right["colC"]]`. This makes you intent more explicit and avoids counting parentheses for long chains of conditions.

Finally, if you are performing an "equi-join", where you are testing for equality between identically named columns, you can just specify the name of the columns as a string or a list of strings as a predicate. In our case, it means that our predicate can only be `"LogServiceID"`. This is what I put in listing 5.4.

Listing 5.4 A barebone recipe for a join in PySpark, with the left and right tables filled in, as well as the predicate.

```
logs.join(
    log_identifier,
    on="LogServiceID"
    how=[METHOD]
)
```

The join method influences how you structure predicates, so XREF ch05-join-naming revisits the whole join operation after we're done with the ingredient-by-ingredient approach. The last parameter is the `how`, which completes our join operation.

5.1.4 How do you do it: the join method

The last ingredient of a successful join is the `how` parameter, which will indicate the join method. Most books explaining joins shows Venn diagrams indicating how each joins colors the different areas, but I find that it is only useful as a reminder, not a teaching tool. I'll review each type of join with the same tables we've used in figure 5.1, giving the result of the operation.

A join method basically boils down to those two questions:

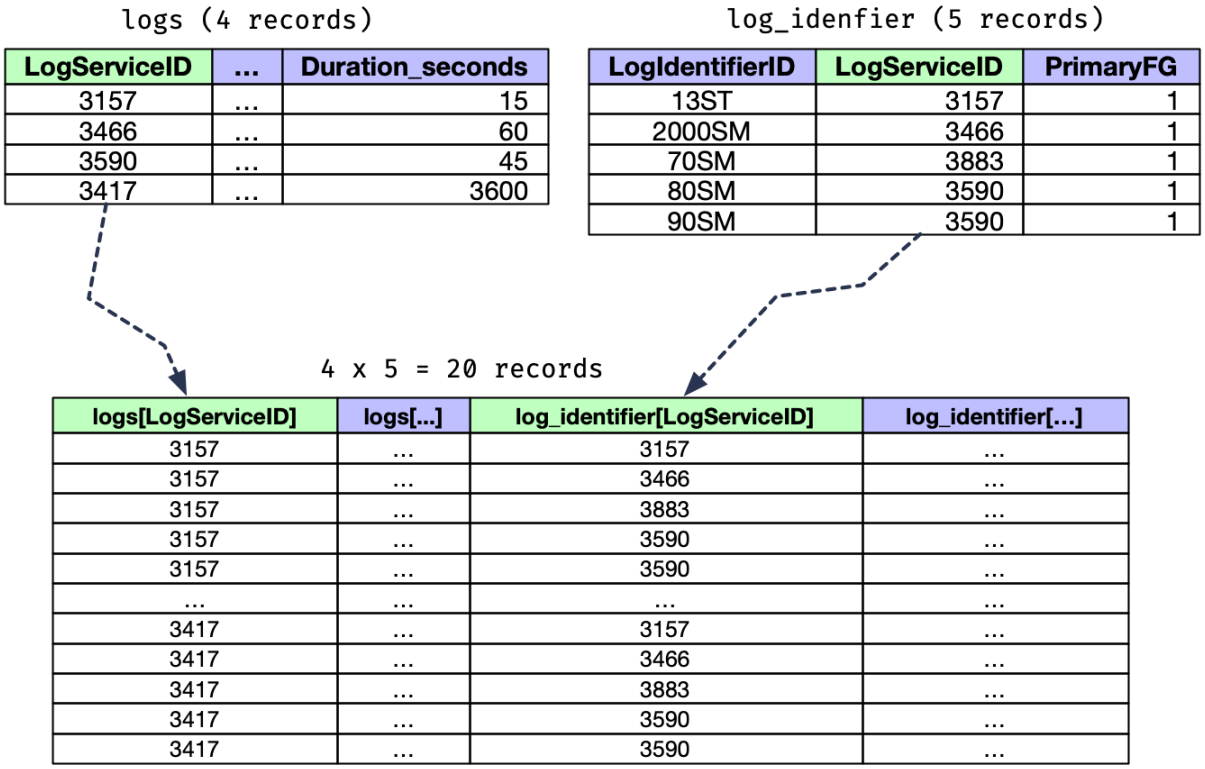
1. What happens when the return value of the predicates is `True`?
2. What happens when the return value of the predicates is `False`?

Classifying the join methods based on the answer to those two questions is an easy way to remember them.

> **TIP** PySpark's joins are basically the same as SQL joins. If you are already comfortable with them, feel free to skip this section.

CROSS JOIN: THE NUCLEAR OPTION

A cross join (`how="cross"`) is the nuclear option. It returns a record for every record pair, *regardless* of the value the predicates return. In our data frame example, our `logs` table contains 4 records and our `logs_identifier` contains 5 records, so the cross join will contain 4 x 5 = 20 records. The result is illustrated in figure 5.2.



LogServiceID	...	Duration_seconds
3157	...	15
3466	...	60
3590	...	45
3417	...	3600

LogIdentifierID	LogServiceID	PrimaryFG
13ST	3157	1
2000SM	3466	1
70SM	3883	1
80SM	3590	1
90SM	3590	1

logs[LogServiceID]	logs[...]	log_identifier[LogServiceID]	log_identifier[...]
3157	...	3157	...
3157	...	3466	...
3157	...	3883	...
3157	...	3590	...
3157	...	3590	...
...	...	...	...
3417	...	3157	...
3417	...	3466	...
3417	...	3883	...
3417	...	3590	...
3417	...	3590	...

Figure 5.2 A visual example of a cross join. Each record on the left is matched to every record on the right.

Cross joins are seldom the operation that you want, but they are useful when you want a table that contains every possible combination.

> **TIP** PySpark also provides an explicit `crossJoin()` method that takes the right data frame as a parameter.

INNER JOIN

An inner join (`how="inner"`) is the most common join by a landslide. PySpark will default to an inner join if you don't pass a join method explicitly. It returns a record if the predicate is true and drops it if false. I consider inner join to be the natural way to think of joins because they are very simple to reason about.

If we look at our tables, we would have a table very similar to figure 5.1. The record with the `LogServiceID == 3590` on the left will be duplicated because it matches two records on the right table. The result is illustrated in figure 5.3.

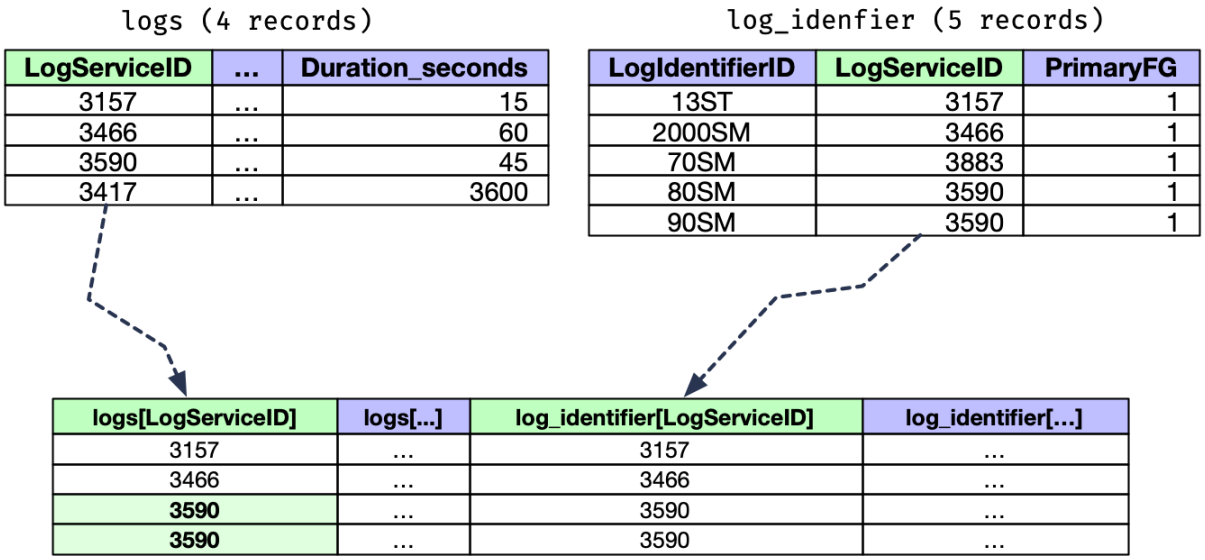


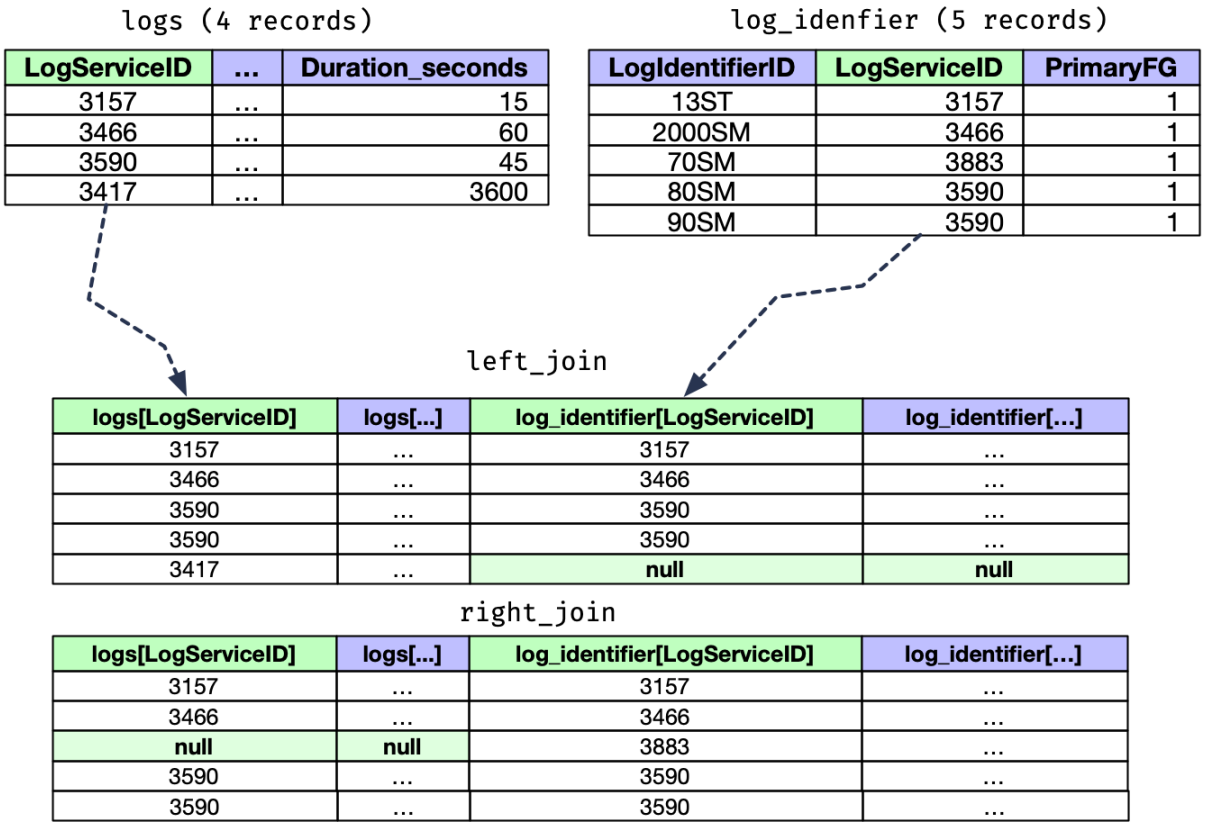
Figure 5.3 An inner join. Each successful predicate creates a joined record.

LEFT AND RIGHT OUTER JOIN

Left (`how="left"` or `how="left_outer"`) and right (`how="right"` or `how="right_outer"`) are like inner join, in which they generate a record for a successful predicate. The difference is what happens when the predicate is false:

- A *left* (also called a *left outer*) join will add the unmatched records from the *left* table in the joined table, filling the columns coming from the *right* table with `None/null`
- A *right* (also called a *right outer*) join will add the unmatched records from the *right* in the joined table, filling the columns coming from the *left* table with `None/null`

In practice, it means that your joined table is guaranteed to contain all the records of the table which feeds the join (left or right). Visually, figure 5.4 shows that, although `3417` doesn't satisfy the predicate, it is still present in the left joined table. The same happens with `3883` and the right table. Just like an inner join, if the predicate is successful more than once, the record will be duplicated.



LogServiceID	...	Duration_seconds
3157	...	15
3466	...	60
3590	...	45
3417	...	3600

LogIdentifierID	LogServiceID	PrimaryFG
13ST	3157	1
2000SM	3466	1
70SM	3883	1
80SM	3590	1
90SM	3590	1

logs[LogServiceID]	logs[...]	log_identifier[LogServiceID]	log_identifier[...]
3157	...	3157	...
3466	...	3466	...
3590	...	3590	...
3590	...	3590	...
3417	...	**null**	**null**

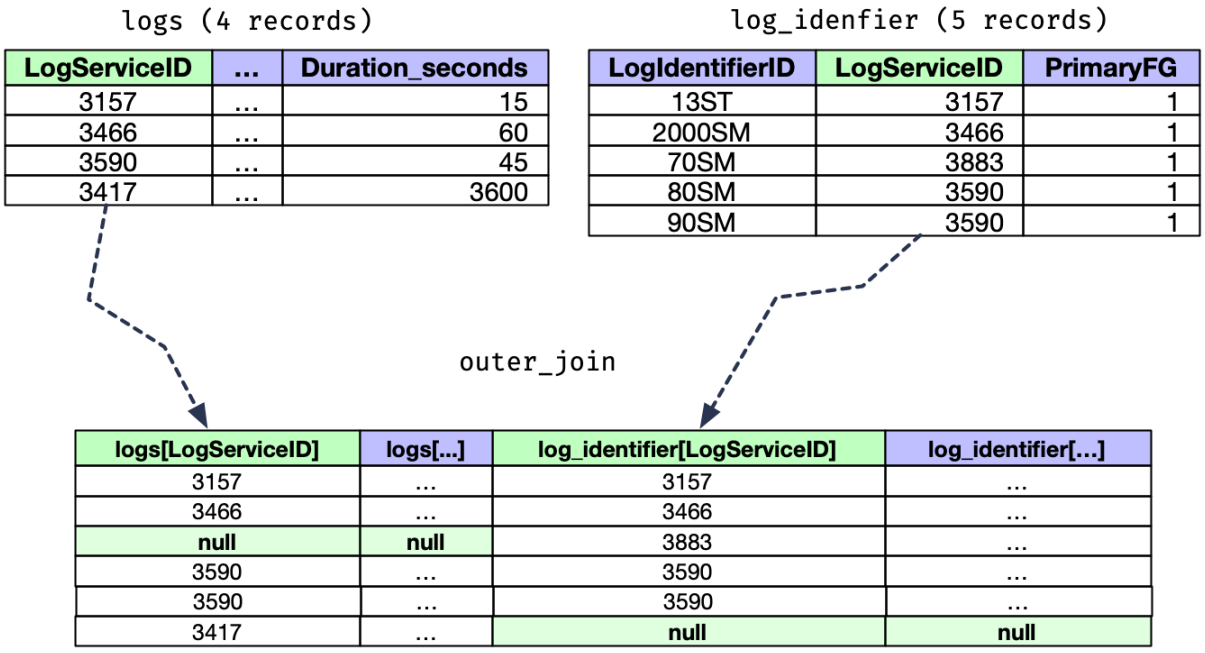
logs[LogServiceID]	logs[...]	log_identifier[LogServiceID]	log_identifier[...]
3157	...	3157	...
3466	...	3466	...
null	**null**	3883	...
3590	...	3590	...
3590	...	3590	...

Figure 5.4 A left and right joined table. All the records of the direction table are present in the resulting table.

Left and right joins are very useful when you are not certain if the link table contains every key. You can then fill the null values— —— see XREF ch05-fillna— —— or process them knowing you didn't drop any records.

FULL OUTER JOIN

A full outer (`how="outer"`, `how="full"` or `how="full_outer"`) join is simply the fusion of a left and right join. It will add the unmatched records from the left and the right table, padding with `None/null`. It serves a similar purpose to the left and right join but is not as popular since you'll generally have one (and just one) anchor table that you want to preserve all records.



LogServiceID	...	Duration_seconds
3157	...	15
3466	...	60
3590	...	45
3417	...	3600

LogIdentifierID	LogServiceID	PrimaryFG
13ST	3157	1
2000SM	3466	1
70SM	3883	1
80SM	3590	1
90SM	3590	1

logs[LogServiceID]	logs[...]	log_identifier[LogServiceID]	log_identifier[...]
3157	...	3157	...
3466	...	3466	...
null	**null**	3883	...
3590	...	3590	...
3590	...	3590	...
3417	...	**null**	**null**

Figure 5.5 A left and right joined table. We can see all the records from both table.

LEFT SEMI-JOIN AND LEFT ANTI-JOIN

The left semi-join and left anti-join are a little more esoteric, but are really easy to understand.

A left semi-join (`how="left_semi"`) is the same as an **inner** join, but only keeps the columns in the left table. It also won't duplicate the records in the left table if they fulfill the predicate with more than one record in the right table. Its main purpose is to filter out records from a table based on a predicate depending of another table.

A left anti-join is the opposite of an **inner** join. It will keep only the records that do not match the predicate with any record in the right table, dropping any successful match.

Our blueprint join is now finalized: we are going with an inner join since we want to keep only the records where the `LogServiceID` has additional information in our `log_identifier` table. Since our join is complete, I assign the result to a new variable `logs_and_channels`.

Listing 5.5 A barebone recipe for a join in PySpark, with the left and right tables filled in, as well as the predicate and the method.

```
logs_and_channels = logs.join(
    log_identifier,
    on="LogServiceID"
    how="inner"    ❶
)
```

❶ I could have omitted the `how` parameter outright, since inner join is the default.

With the first join done, we will link two additional tables to continue our data discovery and

processing. The `CategoryID` table contains information about the type of programs and the `ProgramClassID` table contains the data that allows us to pinpoint the commercials.

This time, we are performing `left` joins since we are not entirely certain about the existence of the keys in the link table. In listing 5.6, we follow the same process as we did for the `log_identifier` table, in one fell swoop.

- We read the table using the `SparkReader.csv` and the same configuration as our other tables.
- We keep the relevant columns.
- We join the data to our `logs_and_channels` table, using PySpark's method chaining.

Listing 5.6 Linking the category and program class tables using a left join

```
DIRECTORY = "../../data/Ch04"

cd_category = spark.read.csv(
    os.path.join(DIRECTORY, "ReferenceTables/CD_Category.csv"),
    sep="|",
    header=True,
    inferSchema=True,
).select(
    "CategoryID",
    "CategoryCD",
    F.col("EnglishDescription").alias("Category_Description"),    ❶
)

cd_program_class = spark.read.csv(
    os.path.join(DIRECTORY, "ReferenceTables/CD_ProgramClass.csv"),
    sep="|",
    header=True,
    inferSchema=True,
).select(
    "ProgramClassID",
    "ProgramClassCD",
    F.col("EnglishDescription").alias("ProgramClass_Description"),    ❷
)

full_log = logs_and_channels.join(cd_category, "CategoryID", how="left").join(
    cd_program_class, "ProgramClassID", how="left"
)
```

❶ We're aliasing the `EnglishDescription` column to remember what it maps to

❷ Same as #1 here, but for the program class.

In listing 5.6, in the select statement, I aliased some columns that were both named `EnglishDescription`. How will PySpark know which one to keep? The next section takes a brief detour from our exploration to address one of the most frequent sources of frustration when dealing with columns: staying on top of you column names when joining data frames.

SIDEBAR **The science of joining in a distributed environment**

When joining data in a distributed environment, the "we don't care about where data is" doesn't work anymore. To be able to process a comparison between records, the data needs to be on the same machine. If not, PySpark will move the data in an operation called a *shuffle*. As you can imagine, moving large amounts of data over the network is very slow, and we should aim to avoid this when possible.

This is one of the instances where PySpark's abstraction model shows some weakness. Since joins are such an important part of working with multiple data sources, I've decided to introduce the syntax here so we can get things rolling. We revisit performance considerations and join strategies in chapter 9. For the moment, trust the optimizer!

5.1.5 Naming conventions in the joining world

By default, PySpark will not allow two columns to be named the same. If you create a column with `withColumn()` using an existing column name, PySpark will overwrite (or shadow) the column. When joining data frames, the situation is a little more complicated, as displayed in listing 5.7

Listing 5.7 A join that generates two seemingly identically named columns

```
logs_and_channels_verbose = logs.join(
    log_identifier, logs["LogServiceID"] == log_identifier["LogServiceID"]
)

logs_and_channels_verbose.printSchema()

# root
#  |-- LogServiceID: integer (nullable = true)   ❶
#  |-- LogDate: timestamp (nullable = true)
#  |-- AudienceTargetAgeID: integer (nullable = true)
#  |-- AudienceTargetEthnicID: integer (nullable = true)
#  [...]
#  |-- duration_seconds: integer (nullable = true)
#  |-- LogIdentifierID: string (nullable = true)
#  |-- LogServiceID: integer (nullable = true)   ❷
#  |-- PrimaryFG: integer (nullable = true)

try:
    logs_and_channels_verbose.select("LogServiceID")
except AnalysisException as err:
    print(err)

# "Reference 'LogServiceID' is ambiguous, could be: LogServiceID, LogServiceID.;"   ❸
```

❶ One `LogServiceID` column…

❷ … and another!

❸ PySpark doesn't know which column we mean.

PySpark happily joins the two data frames together but fails when we try to work with the ambiguous column. This is a frequent situation when working with data that follows the same convention for column naming. Fortunately, solving this problem is easy. I show in this section three methods, from the easiest to the most general.

First, when performing an equi-join, prefer using the simplified syntax, since it takes care of removing the second instance of the predicate column. This only works when using equality comparison, since the data is identical in both columns from the predicate, preventing information loss. I show the code and schema of the resulting data frame when using a simplified equi-join in listing 5.8.

Listing 5.8 Using the simplified syntax for equi-joins results in no duplicate columns.

```
logs_and_channels = logs.join(log_identifier, "LogServiceID")

logs_and_channels.printSchema()

# root
#  |-- LogServiceID: integer (nullable = true)
#  |-- LogDate: timestamp (nullable = true)
#  |-- AudienceTargetAgeID: integer (nullable = true)
#  |-- AudienceTargetEthnicID: integer (nullable = true)
#  |-- CategoryID: integer (nullable = true)
#  [...]
#  |-- Language2: integer (nullable = true)
#  |-- duration_seconds: integer (nullable = true)
#  |-- LogIdentifierID: string (nullable = true)
#  |-- PrimaryFG: integer (nullable = true)
```

The second approach relies on the fact that PySpark joined data frames remembers the provenance of the columns. Because of this, we can refer to the `LogServiceID` columns using the same nomenclature as before, e.g. `log_identifier["LogServiceID"]`. We can then rename this column or delete is, solving our issue. I use this approach in listing 5.9

Listing 5.9 Using the origin name of the column allows for unambiguous selection of the columns after a join

```
logs_and_channels_verbose = logs.join(
    log_identifier, logs["LogServiceID"] == log_identifier["LogServiceID"]
)

logs_and_channels.drop(log_identifier["LogServiceID"]).select("LogServiceID")    ❶

# DataFrame[LogServiceID: int]
```

❶ By dropping one of the two duplicated columns, we can then use the name for the other without any problem.

The last approach is convenient if you use the `Column` object directly. PySpark will not resolve the origin name when you rely on `F.col()` to work with columns. To solve this the most general way, we need to `alias()` our tables when performing the join, as shown in listing 5.10.

Listing 5.10 Aliasing our tables makes the origin resolved when using `F.col()` to refer to columns.

```
logs_and_channels_verbose = logs.alias("left").join(    ❶
    log_identifier.alias("right"),    ❷
    logs["LogServiceID"] == log_identifier["LogServiceID"],
)

logs_and_channels_verbose.drop(F.col("right.LogServiceID")).select(
    "LogServiceID"
)    ❸

# DataFrame[LogServiceID: int]
```

❶ Our `logs` table gets aliased as `left`.

❷ Our `log_identifier` gets aliased as `right`.

❸ `F.col()` will resolve `left` and `right` as prefix for the column names.

All three approaches are valid. The first one works only in the case of equijoins, but the two others are mostly interchangeable. PySpark gives you a lot of control over the structure and naming of your data frame but requires you to be explicit.

This section packed a lot of information about joins, a very important tool when working with interrelated data frames. Although the possibilities are endless, the syntax is simple and easy to understand.

`left.join(right`, who's the first parameter. `on` decides if it's a match. `how` indicates how to operate on match success and failures.

Maybe we could turn this into a rap…

With our table nicely augmented, let's carry on to our last step: summarizing the table using groupings.

SIDEBAR **Exercise 5.1**

What is the result of this code?

one = left.join(right, how="left_semi", on="my_column" two = left.join(right, how="left_anti", on="my_column") one.union(two)

SIDEBAR **Exercise 5.2**

Write a PySpark code that will return the result of the following code block without using a left anti join.

left.join(right, how="left_anti", on="my_column").select("my_column").distinct()

SIDEBAR **Exercise 5.3 (hard)**

Write a PySpark code that will return the result of the following code block without using a left semi-join.

```
left.join(right, how="left_semi",
on="my_column").select("my_column").distinct()
```

5.2 Summarizing the data via: groupby and GroupedData

When displaying data, especially large amounts of data, you'll often summarize data using statistics as a first step. As a matter of fact, chapter 4 showed how you can use `summary()` and `display()` to compute mean, min, max, etc. over the whole data frame. What if we need to look using a different lens?

This section covers the `groupby()` method in greater detail than seen in chapter 3. I introduce here the `GroupedData` object and its usage. In practical terms, we'll use `groupby()` to answer our original question: what are the channels with the most and least proportion of commercials? In order to answer this, we have to take each channel and sum the `duration_seconds` in two ways:

1. One to get the number of seconds when the program is a commercial.
2. One to get the number of seconds of total programming.

Our plan, before we start summing, is to identify what is considered a commercial and what is not. The documentation doesn't provide formal guidance on how to do so, so we'll explore the data and draw our conclusion. Let's group!

5.2.1 A simple groupby blueprint

In chapter 3, we performed a very simple `groupby()` to count the occurrences of each word. It was a very simple example of grouping and counting records based on the words inside the (only) column. In this section, we expand on that simple example by grouping over many columns. I also introduce a more general notation than the `count()` we've used previously, so we compute more that one summary function.

Since you are already acquainted with the basic syntax of `groupby()`, this section starts by presenting a full code block that computes the total duration (in seconds) of the program class. In listing 5.11 we perform the grouping, compute the aggregate function, and present the results in decreasing order.

Listing 5.11 Displaying the most popular types of programs

```
full_log.groupby("ProgramClassCD", "ProgramClass_Description").agg(
    F.sum("duration_seconds").alias("duration_total")
).orderBy("duration_total", ascending=False).show(100, False)

# +--------------+-------------------------------------+--------------+
# |ProgramClassCD|ProgramClass_Description             |duration_total|
# +--------------+-------------------------------------+--------------+
# |PGR           |PROGRAM                              |652802250     |
# |COM           |COMMERCIAL MESSAGE                   |106810189     |
# |PFS           |PROGRAM FIRST SEGMENT                |38817891      |
# |SEG           |SEGMENT OF A PROGRAM                 |34891264      |
# |PRC           |PROMOTION OF UPCOMING CANADIAN PROGRAM|27017583      |
# |PGI           |PROGRAM INFOMERCIAL                  |23196392      |
# |PRO           |PROMOTION OF NON-CANADIAN PROGRAM    |10213461      |
# |OFF           |SCHEDULED OFF AIR TIME PERIOD        |4537071       |
# [... more rows]
# |COR           |CORNERSTONE                          |null          |
# +--------------+-------------------------------------+--------------+
```

This small program has a few new parts, so let's review them one by one.

Our grouping routing starts with the `groupby()` method. A "grouped by" data frame is not a data frame anymore, instead, it becomes a `GroupedData` object, displayed in all its glory in listing 5.13. This object a transitional object: you can't really inspect it (there is no `.show()` method) and it's waiting for further instructions to become show-able again. Illustrated, it would look like the right-hand side of figure 5.7. You have the key (or keys, if you `groupby()` multiple columns), and the rest of the columns are grouped inside some "cell", awaiting a summary function so they can be promoted to a bona fide column again.

SIDEBAR **`agg()` for the lazy**

`agg()` also accepts a dictionary, in the form `{column_name: aggregation_function}` where both are string. Because of this, we can rewrite listing 5.11 like so.

Listing 5.12 Displaying the most popular types of programs, using a dictionary expression inside `agg()`

```
full_log.groupby("ProgramClassCD", "ProgramClass_Description").agg(
    {"duration_seconds": "sum"}
).withColumnRenamed("sum(duration_seconds)", "duration_total").orderBy(
    "duration_total", ascending=False
).show(
    100, False
)
```

It makes rapid prototyping very easy (you can, just like with column objects, use the `"*"` to refer to all columns). I personally don't like this approach for most cases since you don't get to alias your columns when creating them. I am including it since you will see it when reading other people's code.

Listing 5.13 A `GroupedData` object representation. Unlike the data frame, we have no information about the columns or the structure.

```
full_log.groupby()
# <pyspark.sql.group.GroupedData at 0x119baa4e0>
```

logs: DataFrame

ProgramClassCD	ProgramClass_Description	...	Duration_seconds
PGR	PROGRAM	...	15
PGR	PROGRAM	...	60
COM	COMMERCIAL MESSAGE	...	45
COM	COMMERCIAL MESSAGE	...	3600
PGR	PROGRAM	...	30
PFS	PROGRAM FIRST SEGMENT	...	60
...	...	...	540
COM	COMMERCIAL MESSAGE	...	60

Figure 5.6 The original data frame, with the focus on the columns we are grouping by.

logs.groupby("ProgramClassCD", "ProgramClass_Description"): GroupedData

ProgramClassCD	ProgramClass_Description	[Group cell]
PGR	PROGRAM	... | Duration_seconds ... | 15 ... | 60 ... | 30
COM	COMMERCIAL MESSAGE	... | Duration_seconds ... | 45 ... | 3600 ... | 60
PFS	PROGRAM FIRST SEGMENT	... | Duration_seconds ... | 60

There is one record per key set (in this case, each (ProgramClassCD, ProgramClass_Description is unique)

All the non-key columns are here, split between the key columns value.

Figure 5.7 The GroupedData object resulting from grouping by (ProgramClassID, ProgramClass_Description). The non-key columns are all in stand-by in the group cell.

In chapter 3, we brought back the `GroupedData` into a data frame by using the `count()` method, which returns the count of each group. There are a few other, such as `min()`, `max()`, `mean()` or `sum()`. We could have used the `sum()` method directly, but we wouldn't have had the option of aliasing the resulting column, getting stuck with `sum(duration_seconds)` for a name. Instead, we use the oddly named `agg()`.

The `agg()` method, for aggregate (or aggregation?), will take one or more *aggregate functions* from the `pyspark.sql.functions` module we all know and love and apply them on each group of the `GroupedData` object. In figure 5.8, I start on the left with our `GroupedData` oblect. Calling `agg()` with an appropriate aggregate function pulls the column from the group cell, extracts the values, and performs the function, yielding the answer. Compared to using the `sum()` function on the group by object, `agg()` trades a few keystrokes for 2 main advantages:

1. `agg()` takes an arbitrary number of aggregate functions, unlike using a summary method directly. You can't chain multiple functions on `GroupedData` objects: the first one will transform it into a data frame, and the second one will fail.
2. You can alias resulting columns, so you control their name and improve the robustness of your code.

```
logs.groupby(
    "ProgramClassCD", "ProgramClass_Description"
).agg(
    F.sum(F.col("Duration_seconds"))
): DataFrame
```

ProgramClassCD	ProgramClass_Description	Duration_seconds
PGR	PROGRAM	15 + 60 + 30 = 105
COM	COMMERCIAL MESSAGE	45 + 3600 + 60 = 3705
PFS	PROGRAM FIRST SEGMENT	60

Figure 5.8 A data frame arising from the application of the agg() method (aggregate function: F.sum() on Duration_seconds)

After the application of the aggregate function on our `GroupedData` object, we're back with a data frame. We can then use the `orderBy` method to order the data by decreasing order of `duration_total`, our newly created column. We finish by showing 100 rows, which is more than what the data frame contains, so it shows everything.

Let's select our commercials. 5.1 shows my picks.

Table 5.1 The types of programs we'll be considering as commercials

ProgramClassCD	ProgramClass_Description	duration_total
COM	COMMERCIAL MESSAGE	106810189
PRC	PROMOTION OF UPCOMING CANADIAN PROGRAM	27017583
PGI	PROGRAM INFOMERCIAL	23196392
PRO	PROMOTION OF NON-CANADIAN PROGRAM	10213461
LOC	LOCAL ADVERTISING	483042
SPO	SPONSORSHIP MESSAGE	45257
MER	MERCHANDISING	40695
SOL	SOLICITATION MESSAGE	7808

Now that we've done the hard job of identifying our commercial codes, we can start counting!

SIDEBAR **`agg()` is not the only player in town**

Since PySpark 2.3, you can also use `groupby()` with the `apply()` method, in the creatively named "split-apply-combine" pattern. I cover this pattern in chapter 8.

5.2.2 A column is a column: using agg with custom column definitions

When grouping and aggregating columns in PySpark, we have the whole power of the `Column` object at our fingertips. This means that we can group by and aggregate on custom columns! For this section, we will start by building a definition of `duration_commercial`, which takes the duration of a program only if it is a commercial, and use this in our `agg()` statement to seamlessly compute both the total duration and the commercial duration.

If we encode the content of 5.1 into a PySpark definition, this gives us listing 5.14

Listing 5.14 Computing only the commercial time for each program in our table

```
F.when(
    F.trim(F.col("ProgramClassCD")).isin(
        ["COM", "PRC", "PGI", "PRO", "PSA", "MAG", "LOC", "SPO", "MER", "SOL"]
    ),
    F.col("duration_seconds"),
).otherwise(0)
```

I think that the best way to describe the code this time is to literally translate it into plain English.

***When** the field of the **col**(umn) **"ProgramClass"**, **trim**(med) of spaces at the beginning and end of the field **is in** our list of commercial codes, then take the value of the field in the column **"duration_seconds"**. **Otherwise**, use 0 as a value.*

The blueprint of the `F.when()` function is as follows. It is possible to chain multiple `when()` if

we have more than one condition, and to omit the `otherwise()` if we're okay with having `null` values when none of the tests are positive.

(F.when([BOOLEAN TEST], [RESULT IF TRUE]) .when([ANOTHER BOOLEAN TEST], [RESUL .otherwise([DEFAULT RESULT, WILL DEFAULT TO null IF OMITTED]))

We now have a column ready to use. While, we could create the column before grouping by, using `withColumn()`, let's take it up a notch and use our definition directly in the `agg()` clause. Listing 5.15 does just that, and at the same time, gives us our answer!

Listing 5.15 Using our new column into `agg()` to compute our final answer!

```
answer = (
    full_log.groupby("LogIdentifierID")
    .agg(
        F.sum(
            F.when(
                F.trim(F.col("ProgramClassCD")).isin(
                    ["COM", "PRC", "PGI", "PRO", "LOC", "SPO", "MER", "SOL"]
                ),
                F.col("duration_seconds"),
            ).otherwise(0)
        ).alias("duration_commercial"),
        F.sum("duration_seconds").alias("duration_total"),
    )
    .withColumn(
        "commercial_ratio", F.col("duration_commercial") / F.col("duration_total")
    )
)

answer.orderBy("commercial_ratio", ascending=False).show(1000, False)

# +--------------+------------------+--------------+--------------------+
# |LogIdentifierID|duration_commercial|duration_total|commercial_ratio    |
# +--------------+------------------+--------------+--------------------+
# |HPITV         |403               |403           |1.0                 |
# |TLNSP         |234455            |234455        |1.0                 |
# |MSET          |101670            |101670        |1.0                 |
# |TELENO        |545255            |545255        |1.0                 |
# |CIMT          |19935             |19935         |1.0                 |
# |TANG          |271468            |271468        |1.0                 |
# |INVST         |623057            |633659        |0.9832686034602207  |
# [...]
# |OTN3          |0                 |2678400       |0.0                 |
# |PENT          |0                 |2678400       |0.0                 |
# |ATN14         |0                 |2678400       |0.0                 |
# |ATN11         |0                 |2678400       |0.0                 |
# |ZOOM          |0                 |2678400       |0.0                 |
# |EURO          |0                 |null          |null                |
# |NINOS         |0                 |null          |null                |
# +--------------+------------------+--------------+--------------------+
```

Wait a moment? Are some channels *only commercials*? That can't be it. If we look at the total duration, we can see that some channels don't broadcast a lot (1 day = 86,400 seconds). Still, we accomplished our goal: we identified the channels with the most commercials. We finish this chapter with one last task: processing those `null` values.

5.3 Taking care of null values: drop and fill

Null values represent the absence of value. I find this to be a great oxymoron: a value for no value? Philosophy aside, we have some nulls in our result set and I would like them gone.

PySpark provides two main functionalities to deal with null values: you can either `dropna()` the record containing them of `fillna()` the null with a value. We explore in this section both options to see which one is best for our analysis.

5.3.1 Dropping it like it's hot

Our first option would be to plainly ignore the records that have null values. For this, we use the data frame `dropna()` method. It takes three parameters:

1. `how`, which can take the value `any` or `all`. If `any` is selected, PySpark will drop records where *at least one* of the fields are null. In the case of `all`, only the records where all fields are null will be removed. By default, PySpark will take the `any` mode.
2. `thresh` takes an integer value. If set (its default is None), PySpark will ignore the `how` parameter and only drop the records with less than thresh non-null values.
3. Finally, `subset` will take an optional list of columns that drop will use to make its decision.

In our case, we want to keep only the records that have a `commercial_ratio` that is non-null. We just have to pass our column to the `subset` parameter, like in listing 5.16.

Listing 5.16 Dropping only the records that have a `commercial_ratio` value of `null`.

```
answer_no_null = answer.dropna(subset=["commercial_ratio"])

answer_no_null.orderBy("commercial_ratio", ascending=False).show(1000, False)

# +---------------+------------------+--------------+---------------------+
# |LogIdentifierID|duration_commercial|duration_total|commercial_ratio     |
# +---------------+------------------+--------------+---------------------+
# |HPITV          |403               |403           |1.0                  |
# |TLNSP          |234455            |234455        |1.0                  |
# |MSET           |101670            |101670        |1.0                  |
# |TELENO         |545255            |545255        |1.0                  |
# |CIMT           |19935             |19935         |1.0                  |
# |TANG           |271468            |271468        |1.0                  |
# |INVST          |623057            |633659        |0.9832686034602207   |
# [...]
# |OTN3           |0                 |2678400       |0.0                  |
# |PENT           |0                 |2678400       |0.0                  |
# |ATN14          |0                 |2678400       |0.0                  |
# |ATN11          |0                 |2678400       |0.0                  |
# |ZOOM           |0                 |2678400       |0.0                  |
# +---------------+------------------+--------------+---------------------+

print(answer_no_null.count())  # 322
```

This option is completely legitimate, but it removes some records from our data frame. What if we want to keep everything?

5.3.2 Filling values to our heart's content

The yin to `dropna()` yang is to provide a default value to the null values. For this, PySpark provided the `fillna()` method. This method takes two parameters.

1. The `value`, which is either a Python int, float, string or bool. PySpark will only fill the compatible columns: for instance, if we were to `fillna("zero")`, our `commercial_ratio`, being a double, would not be filled.
2. The same `subset` parameter we encountered in `dropna()`. We can limit the scope of our filling to only the columns we want.

Concretely, a `null` value in any of our numerical columns means that the value should be zero, so listing 5.17 fills the null values with 0.

Listing 5.17 Filling our numerical records with 0 using the `fillna()` method.

```
answer_no_null = answer.fillna(0)

answer_no_null.orderBy("commercial_ratio", ascending=False).show(1000, False)

# +---------------+------------------+--------------+--------------------+
# |LogIdentifierID|duration_commercial|duration_total|commercial_ratio    |
# +---------------+------------------+--------------+--------------------+
# |HPITV          |403               |403           |1.0                 |
# |TLNSP          |234455            |234455        |1.0                 |
# |MSET           |101670            |101670        |1.0                 |
# |TELENO         |545255            |545255        |1.0                 |
# |CIMT           |19935             |19935         |1.0                 |
# |TANG           |271468            |271468        |1.0                 |
# |INVST          |623057            |633659        |0.9832686034602207  |
# [...]
# |OTN3           |0                 |2678400       |0.0                 |
# |PENT           |0                 |2678400       |0.0                 |
# |ATN14          |0                 |2678400       |0.0                 |
# |ATN11          |0                 |2678400       |0.0                 |
# |ZOOM           |0                 |2678400       |0.0                 |
# +---------------+------------------+--------------+--------------------+

print(answer_no_null.count())  # 324   ❶
```

❶ We have the two additional records that listing 5.16 dropped.

SIDEBAR **The return of the dict**

You can also pass a dict to the fillna method, with the column names as key and the values as dict values. If we were to use this method for our filling, the code would be like listing 5.18.

Listing 5.18 Filling our numerical records with 0 using the `fillna()` method and a dict.

```
answer_no_null = answer.fillna(
    {"duration_commercial": 0, "duration_total": 0, "commercial_ratio": 0}
)
```

Just like with `agg()`, I prefer avoiding the dict approach because I find it less readable. In this case, you can chain multiple `fillna()` to achieve the same result, with better readability.

Our program now is devoid of null values, and we have a full list of channels and their associated ratio of commercial programming.

5.4 What was our question again: our end-to-end program

At the beginning of the chapter, we gave ourselves an anchor question to start exploring the data and uncover some insights. Through the chapter, we've assembled a cohesive dataset containing the relevant information to identify commercial programs and have ranked the channels based on how much of their programming is commercial. In Listing 5.19, I've assembled all the relevant code blocks introduced in the chapter into a single program you can `spark-submit`. The code is also available in the book's repository, under `src/Ch05/commercials.py`. The end-of-chapter exercises also use this code.

Not counting data ingestion, comments or docstring, our code is a rather small 35 lines of code. We could play code golf (trying to shrink the number of characters as much as we can), but I think we've struck a good balance between terseness and ease of reading. Once again, we haven't paid much attention to the distributed nature of PySpark. Once again, we took a very descriptive view of our problem and translated it into code via PySpark's powerful data frame abstraction and rich function ecosystems.

This chapter is the last chapter of the first part of the book. You are now familiar with the PySpark ecosystem and how you can use its main data structure, the data frame, to ingest and manipulate two very common sources of data, textual and tabular. You know a variety and method and functions that can be applied to data frames and columns and can apply those to your own data problem. You can also leverage the documentation provided through the PySpark docstrings, straight from the PySpark shell.

There is a lot more you can get from the plain data manipulation portion of the book. Because of this, I created appendix F— —— how to use the online PySpark API documentation— —— as a guide to becoming self-reliant using the API documentation. Now that you have a solid understanding of the data model and how to structure simple data manipulation programs, adding new functions to your PySpark quiver will be easy.

The second part of the book builds heavily on what you've learned so far.

- We dig deeper into PySpark's data model and find opportunities to refine our code. We will also look at PySpark's column types, how they bridge to Python's types, and how to use them to improve the reliability of our code.
- We also look at how PySpark modernizes SQL, an influential language for tabular data manipulation, and how you can blend SQL and Python in a single program.
- We look at promoting pure Python code to run in the Spark distributed environment. We formally introduce a lower-level structure, the Resilient Distributed Dataset (RDD) and its row-major model. We also look at User Defined Functions (UDF) as a way to augment the functionality of the data frame.
- Finally, we go beyond rows and columns by using the document-data capacities of PySpark's data frame.

Listing 5.19 Our full program, ordering channels by decreasing proportion of commercials in their airings.

```
"""commercials.py

This program computes the commercial ratio for each channel present in the
dataset.
"""

import os

import pyspark.sql.functions as F
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName(
    "Getting the Canadian TV channels with the highest/lowest proportion of commercials."
).getOrCreate()

spark.sparkContext.setLogLevel("WARN")

################################################################################
# Reading all the relevant data sources
################################################################################

DIRECTORY = "./data/Ch04"

logs = spark.read.csv(
    os.path.join(DIRECTORY, "BroadcastLogs_2018_Q3_M8.CSV"),
    sep="|",
    header=True,
    inferSchema=True,
)

log_identifier = spark.read.csv(
    os.path.join(DIRECTORY, "ReferenceTables/LogIdentifier.csv"),
    sep="|",
    header=True,
    inferSchema=True,
)

cd_category = spark.read.csv(
    os.path.join(DIRECTORY, "ReferenceTables/CD_Category.csv"),
    sep="|",
    header=True,
    inferSchema=True,
).select(
    "CategoryID",
    "CategoryCD",
    F.col("EnglishDescription").alias("Category_Description"),
)

cd_program_class = spark.read.csv(
    "./data/Ch03/ReferenceTables/CD_ProgramClass.csv",
    sep="|",
    header=True,
    inferSchema=True,
).select(
    "ProgramClassID",
    "ProgramClassCD",
    F.col("EnglishDescription").alias("ProgramClass_Description"),
)

################################################################################
# Data processing
################################################################################

logs = logs.drop("BroadcastLogID", "SequenceNO")
```

```
logs = logs.withColumn(
    "duration_seconds",
    (
        F.col("Duration").substr(1, 2).cast("int") * 60 * 60
        + F.col("Duration").substr(4, 2).cast("int") * 60
        + F.col("Duration").substr(7, 2).cast("int")
    ),
)

log_identifier = log_identifier.where(F.col("PrimaryFG") == 1)

logs_and_channels = logs.join(log_identifier, "LogServiceID")

full_log = logs_and_channels.join(cd_category, "CategoryID", how="left").join(
    cd_program_class, "ProgramClassID", how="left"
)

answer = (
    full_log.groupby("LogIdentifierID")
    .agg(
        F.sum(
            F.when(
                F.trim(F.col("ProgramClassCD")).isin(
                    ["COM", "PRC", "PGI", "PRO", "LOC", "SPO", "MER", "SOL"]
                ),
                F.col("duration_seconds"),
            ).otherwise(0)
        ).alias("duration_commercial"),
        F.sum("duration_seconds").alias("duration_total"),
    )
    .withColumn(
        "commercial_ratio", F.col("duration_commercial") / F.col("duration_total")
    )
    .fillna(0)
)

answer.orderBy("commercial_ratio", ascending=False).show(1000, False)
```

5.5 Summary

- PySpark implements seven join functionalities, using the common "what, on what, how?" questions: cross, inner, left, right, full, left semi and left anti. Choosing the right join method depends on how to process the records that resolve the predicates and the ones that do not.
- PySpark keeps lineage information when joining data frames. Using this information, we can avoid column naming clashes.
- You can group similar values together using the `groupby()` method on a data frame. The method takes a number of column objects or strings representing columns and returns a `GroupedData` object.
- `GroupedData` objects are transitional structures. They contain two types of columns: the *key* columns, which are the one you grouped by with, and the *group cell*, which is a container for all the other columns. In order to return to a data frame, the most common way is to summarize the values in the column via the `agg()` function, or via one of the direct aggregation methods, such as `count()` or `min()`.

5.6 Exercises

5.6.1 Exercise 5.4

Using the data from the `data/Ch04/Call_Signs.csv` (not a typo)— ——— careful: the delimiter here is the comma, not the pipe!— ——— add the `Undertaking_Name` to our final table to display a human-readable description of the channel.

5.6.2 Exercise 5.5

The government of Canada is asking for your analysis, but they'd like the `PRC` to be weighted differently. They'd like each `PRC` second to be considered 0.75 commercial second. Modify the program to account for this change.

5.6.3 Exercise 5.6

On the data frame returned from `commercials.py`, return the percentage of channels in each bucket based on their `commercial_ratio`. (Hint: look at the documentation for `round` in how to truncate a value.)

commercial_ratio	proportion_of_channels
1.0	
0.9	
0.8	
…	
0.1	
0.0	

6 Multi-dimensional data frames: using PySpark with JSON data

This chapter covers:

- How PySpark encodes pieces of data inside columns, and how their type conveys meaning about what operations you can perform on a given column.
- What kind of types PySpark provides, and how they relate to Python's type definition.
- How PySpark can represent multi-dimensional data using compound types.
- How PySpark structures columns inside a data frame, and how you can provide a schema to manage said structure.
- How to transform the type of a column and what are the implications of doing so.
- How PySpark treats null values and how you can work with them.

Data is beautiful.

We give data physical qualities like "beautiful", "tidy" or "ugly", but it doesn't have the same definition there as it would have for a physical object. The same aspect applies to the concept of "data quality": what makes a high-quality data set?

This Chapter will focus on bringing meaning to the data you ingest and process. While we can't always explain everything in data just by looking at it, just peeking at how some data is represented can lay the foundation of a successful data product. We will look at how PySpark organizes data within a data frame to accommodate a wide variety of use-cases. We'll talk about data representation though types, how they can guide our operations and how to avoid common mistakes when working with then.

This Chapter will give you a solid foundation to ingest various data sources and have a head-start at cleaning and giving context to your data. Data cleaning is one of the dark arts of data science and nothing quite replaces experience; knowing how your toolset can support your exploration and will make it easier to go from a messy data set to a useful one. In terms of actual steps

performed, data cleaning is most-similar to data manipulation: everything you've learned in Part 1 will be put to good use, and we'll continue the discovery in the next chapters.

I will approach this Chapter a little differently than the ones in Part 1, as there will not be a main data set and problem statement that will follow us along. Since we want to see the results of our actions, working with small, simple data frames will make it easier to understand the behaviour of PySpark.

For this Chapter and as a convention for the rest of the book, we will be using the following qualified imports. We saw `pyspark.sql.functions` in Part 1, and we will be adding `pyspark.sql.types` to the party. It is common PySpark practice to alias them to `F` and `T` respectively. I like to have them capitalized, so it's obvious what comes from PySpark and what comes from a regular Python module import (which I keep lower-case).

```
import pyspark.sql.functions as F
import pyspark.sql.types as T
```

6.1 Open sesame: what does your data tell you?

Data in itself is not remarkably interesting. It's what we're able to do with it that makes the eyes of data-driven people sparkle. Whether you're developing a high-performing ETL (extract, transform and load) pipeline for your employer or spending a late night fitting a machine learning model, the first step is always to understand the data at hand. Anybody who worked in data can attest: this is easier said than done. Because it's really easy to access data, people and organizations tend to focus most of their efforts collecting and accumulating data, and relatively less time organizing and documenting it.

When working with data, and PySpark is no exception, I consider that there are three layers of understanding data efficiently. Here they are, in order of increasing complexity:

- At the lower level, there are **types**, which gives us guidance on what individual pieces of data stored mean in terms of value and the operations we can and cannot perform. PySpark use types on each column;
- Then there is **structure**, which is how we organize columns within a data frame for both easy and efficient processing;
- Finally, at the top level, we find **semantic**, which uses and goes beyond types and structure. It answers the question about *what* the data is about and *how* we can use it, based on the content. This is usually answered by data documentation and governance principles.

This chapter will focus on the first two aspects, which are types and structure, with a little bit of exploration about the semantic layer and how PySpark treats null values. Deriving full semantic information from your data in context-dependent, and we will cover it through each use-case in the book. If we look at the use cases we've encountered so far, we've approached the semantic layer in two different ways. In practice, and this chapter is no exception, those three concepts are

intertwined. At the risk of selling a punch, demonstrate that the data frame's structural backbone is also a type! Nonetheless, thinking about type, structure and semantic separately is a useful thought exercise to understand what our data is, how it is organized and what it means.

Chapters 2 and 3— counting word occurrences from a body of text— was a pretty simple example from a data understanding point of view. Each line of text became a string of characters (type) in a PySpark data frame (structure), which we tokenized as words (semantic). Those words were cleaned using heuristics based on a rudimentary understanding of the English language before being lower-cased and grouped together. We didn't need much more in terms of documentation or support material as the data organization and meaning were self-explanatory.

Chapters 4 and 5— exploring the proportion of commercial time versus total air time (semantic)— was inputted in tabular form (structure) with mostly numerical and text formats (types). To interpret accurately each code in the table, and how they relate to one another, it required access to a data dictionary (semantic). The government of Canada thankfully provided good enough documentation for us to understand what each code meant and to be able to filter out useless records.

The next chapters will have different data sources, each with their way of forming a semantic layer. As we saw during Part 1, this is "part of the job" and we often don't think about it. It's just when we don't understand what we have in front of you that documentation feels missed. PySpark won't make your incomprehensible data crystal clear by itself, but having a solid foundation with the right type and structure will avoid some headache.

SIDEBAR **The structure of unstructured data**

Chapter 2 tackled the ingestion and processing of what is called *unstructured* data. Isn't it an oxymoron to talk about structure in this case?

If we take a step back and recall the problem at hand, we wanted to count work occurrences from one (or many) text files. The data started unstructured while it was in a bunch of text files, but we gave it structure (a data frame with 1 column, containing one line of text per cell) before starting the transformation.

Because of the data frame abstraction, we implicitly imposed structure to our text collection to be able to work with it. A PySpark data frame can be very flexible into what the cells can contain but wraps any kind of data it ingests into the `Column` and `Row` abstraction, which we cover in . We'll see in Chapter 9 how the RDD approaches structure differently, where it shines, and how we can move from one to the other easily. (Spoiler alert: the data frame is most often the easiest way to go)

6.2 The first step in understanding our data: PySpark's scalar types

When manipulating large data sets, we lose the privilege of examining each record one by one to derive the meaning of our data set. In Part 1, you already learned a few handy methods and how to work with them to display either a sample of the data or a summary of it.

`.show(n)` will display *n* records in easy-to-read format. Passing the method without a parameter will default to 20 records.

`describe(*cols)` will perform and display basic statistics (non-null count, mean, standard deviation, min, and max) for numeric and string columns.

`summary(*statistics)` will perform the statistics passed as a parameter (which you can find in the `pyspark.sql.functions` module) for all numeric and string columns. If you don't pass parameters to the method, it will compute the same statistics as `describe`, with the addition of the approximate quartiles (25%, 50% and 75%).

One aspect we didn't pay much attention is when introducing those functions is the fact that they only operate on numerical and string data. When you take a step back and reflect on this, it makes sense: to compute statistics, you need to be able to apply them to your values. One way that PySpark maintains order in the realm of possible and impossible operations is via the usage of types.

A *type* is, simply put, information that allows the computer to encode and decode data from human representation to computer notation. At the core, computers only understand bits (which are represented in human notation by zeroes and ones). In itself, `01100101` doesn't mean anything. If you give additional context— let's say we're dealing with an integer here— the computer can translate this binary representation to a human-understandable value, here 101. If we were to give this binary representation a string type, this would lead to the letter `e` in ASCII and UTF-8. Knowing the binary representation of data types is not necessary for using PySpark's types efficiently. The most important thing to remember is that it's necessary to understand what kind of data your data frame contains in order to perform the right transformations. Each column has one and only one type associated to it.

The type information is available when you print the name of the data frame by itself. It can also be displayed in a much friendlier form though the `printSchema()` method. We've used this method to reflect on the composition of our data frames all through Part 1, and this will keep on going. This will give you each column name, its type, and if it accepts null values or not (which we'll cover in). Let's take a very simple data frame to demonstrate four different categories of types. The ingestion and schema display happens in . We are again relying on PySpark's schema inference capabilities, using `inferSchema=True`. This parameter, when set to `True`, tells

PySpark to go over the data twice: the first time to infer the types of each column, and the second time to ingest the data according to the inferred type. This effectively make sure you won't get any type mismatch— which can happen when you only sample a subset of records to determine the type— at the expense of a slower ingestion.

Listing 6.1 Ingesting a simple data frame with 4 columns of different type, string, integer, double and date/timestamp

```
$ cat ./data/Ch05/sample_frame.csv

# string_column,integer_column,float_column,date_column
# blue,3,2.99,2019-04-01
# green,7,6.54,2019-07-18
# yellow,10,9.78,1984-07-14
# red,5,5.17,2020-01-01

sample_frame = spark.read.csv(
    "../../data/Ch06/sample_frame.csv", inferSchema=True, header=True
)

sample_frame.show()
# +-------------+--------------+------------+-------------------+
# |string_column|integer_column|float_column|        date_column|
# +-------------+--------------+------------+-------------------+
# |         blue|             3|        2.99|2019-04-01 00:00:00|
# |        green|             7|        6.54|2019-07-18 00:00:00|
# |       yellow|            10|        9.78|1984-07-14 00:00:00|
# |          red|             5|        5.17|2020-01-01 00:00:00|
# +-------------+--------------+------------+-------------------+

sample_frame.printSchema()
# root
#  |-- string_column: string (nullable = true)   ❶
#  |-- integer_column: integer (nullable = true)   ❷
#  |-- float_column: double (nullable = true)   ❸
#  |-- date_column: timestamp (nullable = true)   ❹
```

❶ `string_column` was inferred as being a `string` type (`StringType()`,)

❷ `integer_column` was inferred as being an `integer` type` (`IntegerType()`,)

❸ `float_column` was inferred as being a `double` type (`DoubleType()`,)

❹ `date_column` was inferred as being a `timestamp` type (`TimestampType()`)

PySpark provides a type taxonomy that shares a lot of similarity to Python, with some more granular types for performance and compatibility with other languages Spark speaks. Both and provide a summary of PySpark's types and how they relate to Python's types. Both table and image are also available in the book's GitHub repository, should you want to keep a copy handy.

PySpark's types constructors, which you can recognize easily by their form (`ElementType()`, where `Element` will be the type used) live under the `pyspark.sql.types` module. In , I provide both the type constructor, as well as the string short-hand which will be especially useful in Chapter 7, but can also be used in place of the type constructor. Both will be explained and used from to .

Table 6.1 A summary of the types in PySpark. A star means there are loss of precision.

Type Constructor	String representation	Python equivalent
`NullType()`	`null`	`None`
`StringType()`	`string`	Python's regular strings
`BinaryType()`	N/A	bytearray
`BooleanType()`	`boolean`	bool
`DateType()`	`date`	`datetime.date` (from the `datetime` library)
`TimestampType()`	`timestamp`	`datetime.datetime` (from the `datetime` library)
`DecimalType(p,s)`	`decimal`	`decimal.Decimal` (from the `decimal` library)*
`DoubleType()`	`double`	float
`FloatType()`	`float`	float*
`ByteType()`	`byte` or `tinyint`	int*
`IntegerType()`	`int`	int*
`LongType()`	`long` or `bigint`	int*
`ShortType()`	`short` or `smallint`	int*
`ArrayType(T)`	N/A	list, tuple or Numpy array (from the `numpy` library)
`MapType(K, V)`	N/A	dict
`StructType([…])`	N/A	list or tuple

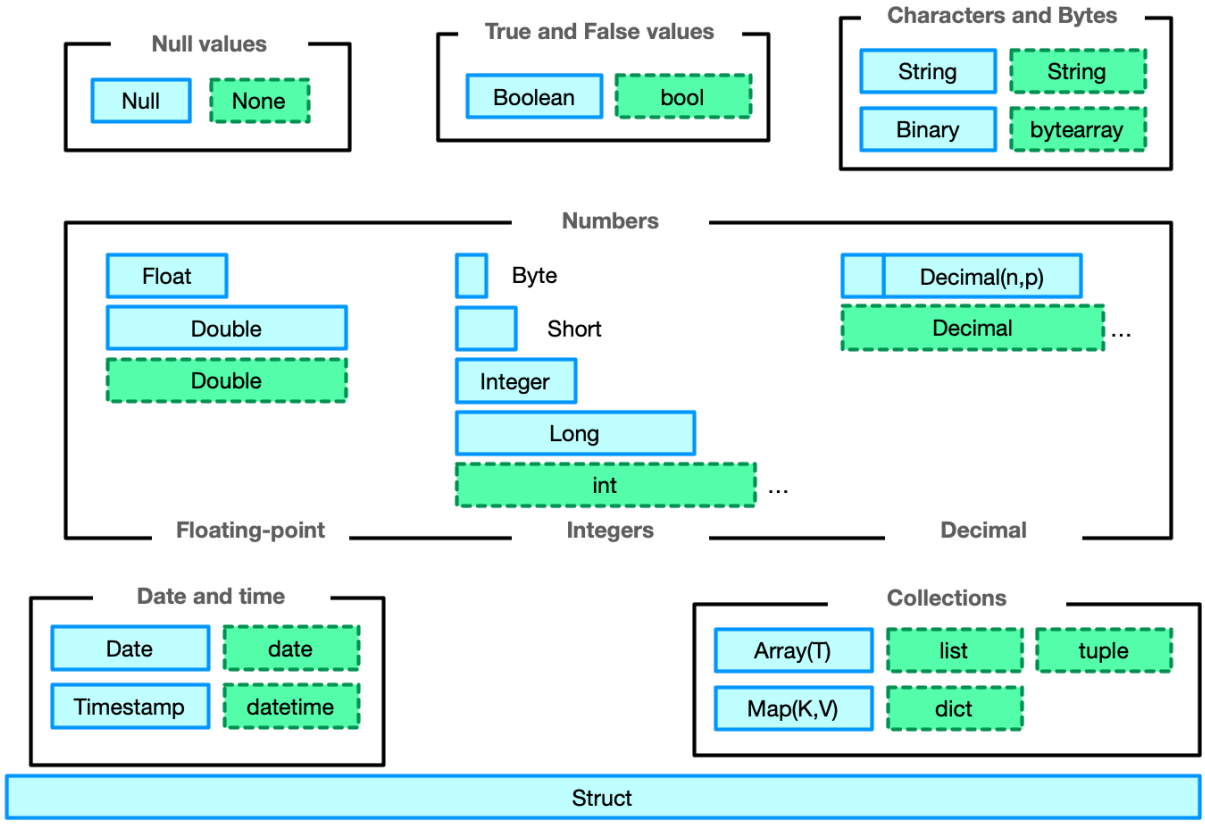


Figure 6.1 An illustrated summary of PySpark's types taxonomy, with Python equivalents

Because some types are very similar, I grouped them into logical entities, based on their behavior. We'll start with string and bytes, before going through numerical values. Date and time structures will follow, before null and boolean. We'll finish by introducing *complex* structures: the array and the map.

SIDEBAR **The eternal fight between Python and the JVM**

Since Spark is a Scala framework first and foremost, a lot of the type conventions comes from how Scala represents types. This has one main advantage: you can move from Spark using another language to PySpark without having to worry about the introduction of type-related bugs. When working on a platform that spans multiple languages, this consistency is really important.

In Chapter 8, when we discuss about creating user-defined functions (UDF), the situation will get a little more complicated, as UDF use Python-defined types. We will at this point discuss strategies to avoid running into the trap of type-mismatch.

6.2.1 String and bytes

Strings and bytes are probably the easiest types to reason about, as they map one-to-one to Python types. A PySpark string can be reasoned just like a Python string. For information that is better represented in a binary format, such as images, videos, and sounds, the `BinaryType()` is very well suited and is analogous to the `bytearray` in Python, which serves the same purpose. We already used `StringType()` and will continue using it through this book. Binary columns representation is a little less ubiquitous, but we see it in action in Chapter 11 when I introduce deep learning on Spark.

There are many things you can do with `String` columns in PySpark. So much, in fact, that by default, PySpark provides only a handful of functions on them. This is where UDF (covered in Chapter 8) will become incredibly useful. A couple interesting functions are provided in `pyspark.sql.functions` to encode and decode string information, such as string-represented JSON (Chapter 10), date or datetime values (Section), or numerical values (). You also get a function to concatenate multiple string or binary columns together (`concat()`, or `concat_ws()` if you want a delimiter between string columns only), compute their length (`length()`).

TIP By default, Spark stores strings using the UTF-8 encoding. This is a sane default for many operations, and is consistent with Python's internal representation too.

Listing 6.2 A simplistic example of string manipulation

```
import pyspark.sql.functions as F

fruits = [
    ["Apple", "Pomme", ""],
    ["Banana", "Banane", ""],
    ["Grape", "Raisin", ""],
]

df = spark.createDataFrame(fruits, ["English", "French", "Russian"])

df.printSchema()

# root
# |-- English: string (nullable = true)
# |-- French: string (nullable = true)
# |-- Russian: string (nullable = true)

df.show()

# +-------+------+--------+
# |English|French| Russian|
# +-------+------+--------+
# |  Apple| Pomme|  |
# | Banana|Banane|   |
# |  Grape|Raisin||
# +-------+------+--------+


df = df.withColumn(
    "altogether", F.concat(F.col("English"), F.col("French"), F.col("Russian"))
)


df = df.withColumn(
    "binary_encoded", F.encode(F.col("altogether"), "UTF-8")    ❶
)
df.select("altogether", "binary_encoded").show(10, False)

# +------------------+-----------------------------------------------------------------------------------------
# |altogether        |binary_encoded
# +------------------+-----------------------------------------------------------------------------------------
# |ApplePomme   |[41 70 70 6C 65 50 6F 6D 6D 65 D1 8F D0 B1 D0 BB D0 BE D0 BA D0 BE]               |
# |BananaBanane  |[42 61 6E 61 6E 61 42 61 6E 61 6E 65 D0 91 D0 B0 D0 BD D0 B0 D0 BD]              |
# |GrapeRaisin|[47 72 61 70 65 52 61 69 73 69 6E D0 B2 D0 B8 D0 BD D0 BE D0 B3 D1 80 D0 B0 D0 B4]|
# +------------------+-----------------------------------------------------------------------------------------

df.withColumn(
    "binary_encoded", F.encode(F.col("altogether"), "UTF-8")
).withColumn("length_string", F.length(F.col("altogether"))).withColumn(
    "length_binary", F.length(F.col("binary_encoded"))
).select(
    "altogether", "binary_encoded", "length_string", "length_binary"
).show(
    10
)

# +------------------+--------------------+-------------+-------------+
# |         altogether|      binary_encoded|length_string|length_binary|
# +------------------+--------------------+-------------+-------------+
# |   ApplePomme|[41 70 70 6C 65 5...|           16|           22|   ❷
# |  BananaBanane|[42 61 6E 61 6E 6...|           17|           22|
# |GrapeRaisin|[47 72 61 70 65 5...|           19|           27|
# +------------------+--------------------+-------------+-------------+
```

❶ We get the raw bytes from the UTF-8 encoded strings in column `together`

❷ `length_string` and `length_binary` return different values since a character (here, Cyrillic characters) can be encoded in more than 1 byte in UTF-8.

6.2.2 The numerical tower(s): integer values

Computer representation of numbers is a fascinating and complicated subject. Besides the fact that we count in base 10, while a computer is fundamentally a base 2 system, many trade-offs are happening between speed, memory usage, precision, and scale. Unsurprisingly, PySpark gives a lot of flexibility in how you want to store your numerical values.

Python, in version 3, streamlined its numeric types and made it much more user-friendly. Decimal values (such as 14.8) are now called `float` and are stored as double-precision floating-point numbers. Integer values (with no decimal values, such as 3 or -349,257,234) are stored as `int` values. Numerical values can take as much memory as necessary in Python: the runtime of the language will allocate enough memory to store your value until you run out of RAM. For more specialized applications, Python also provides rationals (or fractions) and the `Decimal` type, for when you can't afford the loss of precision happening with floating-point numbers. If you are interested in this, Appendix D has a portion about the differences between `float` and `Decimal`.

PySpark follows Spark's convention and provides a much more granular numerical tower. You have more control over memory consumption of your data frame, at the expense of keeping track of the limitations of your chosen type.

We have three main representations of numerical values in PySpark.

1. Integer values, which encompass `Byte`, `Short`, `Integer` and `Long`
2. Floating-point values, which encompass `Float` and `Double`
3. Decimal values, which contains only `Decimal`

Rationals aren't available as a type in PySpark.

The definition of integer, floating-point, and decimal values are akin to what we expect from Python. PySpark provides multiple types for integer and floating-point: the difference between all those types is how many bytes are being used to represent the data in memory. More bytes means a larger span of numbers representable (in the case of integer) or more precision possible (for floating-point numbers). Fewer bytes means a smaller memory footprint which, when you're dealing with a huge number of records, can make a difference in memory usage. With those different types, you have more control regarding how much memory will each value occupy. summarizes the boundaries for integer.

Table 6.2 Integer representation in PySpark: memory footprint, maximum and minimum values

Type Constructor	Number of bits used	Min value (inclusive)	Max value (inclusive)
`ByteType()`	8	-127	128
`ShortType()`	16	-32768	32767
`IntegerType()`	32	2,147,483,648	2,147,483,647
`LongType()`	64	-9,223,372,036,854,775,808	9,223,372,036,854,775,807

Since Python's integers are limited only by the memory of your system, there is always a possible loss of information available when reading very large (both positive or negative) integer values. By default, PySpark will default to a `LongType()` to this problem as much as possible.

Compared to Python, working with a more granular integer tower can cause some headaches at times. PySpark helps a little in that department by allowing numerical types to be mixed and matched when performing arithmetic. The resulting type will be determined by the input types:

1. If both types are identical, the return type will be the input types (`float` + `float` = `float`)
2. If both types are integral, PySpark will return the largest of the two (`int` + `long` = `long`)
3. `Decimal` + Integral Type = `Decimal`
4. `Double` + Any type = `Double`
5. `Float` + Any type (besides `Double`) = `Float`

Remembering all this from the start isn't practical nor convenient. It's best to remember what the types mean and know that PySpark will accommodate the largest memory footprint, without making assumptions on the data. Since floats and doubles are less precise than integers and decimals, PySpark will use those types the moment it encounters them.

When working with bounded numerical types (which means types that can only represent a bounded interval of values), an important aspect to remember is *overflow* and *underflow*. Overflow is when you're storing a value too big for the box you want to put it into. Underflow is the opposite, where you're storing a value too small (too large negatively) for the box it's meant to go into. There are two distinct cases where overflow can happen and both exhibit different behaviour.

The first one is when you're performing an arithmetic operation on integral values that results them overflowing the box they're into. displays what happens when you multiply two shorts (bounded from -32,768 to 32,767) together.

Listing 6.3 Overflowing `Short` values in a data frame leads to a promotion to the appropriate integer type.

```
short_values = [[404, 1926], [530, 2047]]
columns = ["columnA", "columnB"]

short_df = spark.createDataFrame(short_values, columns)

short_df = short_df.select(
    *[F.col(column).cast(T.ShortType()) for column in columns]
)    ❶

short_df.printSchema()

# root
# |-- columnA: short (nullable = true)
# |-- columnB: short (nullable = true)


short_df.show()

# +-------+-------+
# |columnA|columnB|
# +-------+-------+
# |    404|   1926|
# |    530|   2047|   ❷
# +-------+-------+

short_df = short_df.withColumn("overflow", F.col("columnA") * F.col("columnB"))

short_df
# DataFrame[columnA: smallint, columnB: smallint, overflow: smallint]

short_df.show()

# +-------+-------+--------+
# |columnA|columnB|overflow|
# +-------+-------+--------+
# |    404|   1926|   -8328|
# |    530|   2047|  -29202|   ❸
# +-------+-------+--------+
```

❶ Since PySpark defaults integer casting to `Long`, we're re-casting every column of our data frame into `Short` using `select` and a list comprehension over the columns.

❷ Our values are comfortably within the range of shorts.

❸ Multiplying 404 and 1,926 leads to -8,328, and multiplying 530 and 2,047 leads to -29,202 when using short values.

PySpark's default behaviour when dealing with overflow or underflow is to *roll over* the values. Knowing how rolling-over works isn't necessary to productively use PySpark unless you're counting on that behaviour for your application.

SIDEBAR **Rolling over: how integer values are represented in memory**

If you've never had the opportunity to explore how a computer represents integers in memory, you might find the overflow behaviour a little hard to understand. Here is a quick explanation of how integers are stored and what happen when you cast a column into a type too small to contain a given value.

Spark represents integer values using *signed* integers. The first bit is the sign bit and has special behaviour. When set to 1, it will *substract* two to the power of the number of remaining bits (7 in the case of a `Byte`, 15 in the case of a `Short`, and so on). This is how you can get negative values. The rest of the bits are, when set to 1, adding two to the power of their rank, which starts at n-2 (where n is the number of bits) to 0. (An easier way to do it is to start in reverse and start from 0 and increase the exponent). shows how the value 101 is represented in an 8-bit integer value.

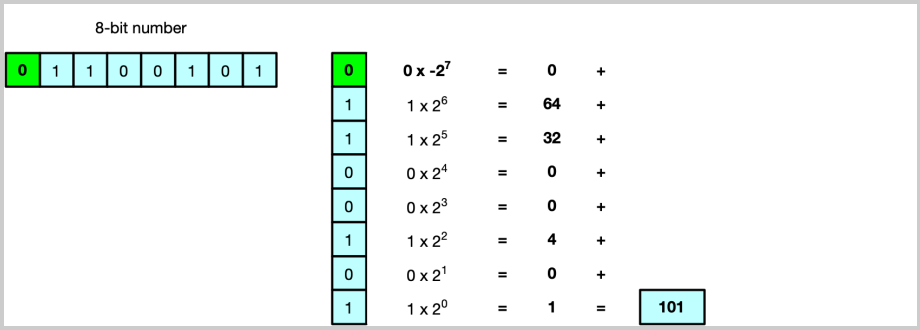


Figure 6.2 A representation of a 8-bit integer storing the value 101 (ByteType)

Taking the first record in , 404 is `00000001 10010100` when represented with 16 bits and 1,926 is `00000111 10000110`. Multiplying those two values yields 778,104, which takes more than 16 bits to represent (`00000000 00001011 11011111 01111000` when using 32 bits). PySpark will keep only the last 16 bits, ignoring the first ones. Our result will then be `11011111 01111000`, or -8,328.

Integer overflow and underflow happens in many programming languages, such as Scala, Java and C and can be the source of hard-to-find bugs. Integer overflow checking adds some overhead that isn't always necessary or wanted, so PySpark leaves this in the control of the developer.

The second time where you can be bitten by integer overflow/underflow is when you're ingesting extremely large values. PySpark, when inferring the schema of your data, will default

integer values to `Long`. What happens when you're trying to read something beyond the bounds of a `Long`? The code in shows what happens in practice when reading large integers.

Listing 6.4 Ingesting numerical values in PySpark

```
df_numerical = spark.createDataFrame(
    [[2 ** 16, 2 ** 33, 2 ** 65]],
    schema=["sixteen", "thirty-three", "sixty-five"],
)

df_numerical.printSchema()
# root
# |-- sixteen: long (nullable = true)
# |-- thirty-three: long (nullable = true)
# |-- sixty-five: long (nullable = true)

df_numerical.show()
# +-------+------------+----------+
# |sixteen|thirty-three|sixty-five|
# +-------+------------+----------+
# |  65536|  8589934592|      null|
# +-------+------------+----------+
```

PySpark refuses to read the value, and silently defaults to a `null` value. If you want to deal with larger numbers than what `Long` provides, `Double` or `Decimal` () will allow for larger numbers. We will also see in how you can take control over the schema of your data frame and set your schema at ingestion time.

6.2.3 The numerical tower(s): double, floats and decimals

Floats have a similar story, with the added complexity of managing scale (what is the range of numbers you can represent) and precision (how many digits after the decimal point can you represent).

- A **single** precision number will encode both integer and decimal part in a 32-bits value.
- A **double** precision number will encode both integer and decimal part in a 64-bits value (double the bits of a single).

PySpark will play nice here and default to `DoubleType()` when reading decimal values, which provides a behavior identical to Python. Some specific applications, such as specialized machine learning libraries will work faster with single-precision numbers. Encoding arbitrary-precision numbers in a binary format is a fascinating subject. Because Python defaults to doubles when working with floating-point values, do the same unless you're short on memory or want to squeeze the most performance out of your Spark cluster.

`Floats` and `Double` are amazing on their own. A `Double` can represent a value up to 1.8×10^{308}, which is much more than a `Long`. One caveat of this number format is *precision*. Because they use base 2 (or binary) representation, some numbers can't be perfectly represented. shows a

simple example using floats and double. In both cases, adding and subtracting the same value leads to a slight imprecision. The `Double` result is much more precise as we can see, and since it follows Python's conventions, I usually prefer it.

Listing 6.5 Demonstrating the precision of double and floats

```
doubles = spark.createDataFrame([[0.1, 0.2]], ["zero_one", "zero_two"])
doubles.withColumn(
    "zero_three", F.col("zero_one") + F.col("zero_two") - F.col("zero_one")
).show()

# +--------+--------+-------------------+
# |zero_one|zero_two|         zero_three|
# +--------+--------+-------------------+
# |     0.1|     0.2|0.20000000000000004|
# +--------+--------+-------------------+

floats = doubles.select([F.col(i).cast(T.FloatType()) for i in doubles.columns])
floats.withColumn(
    "zero_three", F.col("zero_one") + F.col("zero_two") - F.col("zero_one")
).show()

# +--------+--------+----------+
# |zero_one|zero_two|zero_three|
# +--------+--------+----------+
# |     0.1|     0.2|0.20000002|
# +--------+--------+----------+
```

Finally, there is a wild card type: the `DecimalType()`. Where integers and floating-point values are defined by the number of bits (base 2) they use for encoding values, the decimal format uses the same base 10 we use currently. It trades computation performance for a representation we're more familiar with. This is often unnecessary for most applications, but if you're dealing with money or values where you can't afford to be even a little bit off (such as money calculations), then `Decimal` will be worth it.

`Decimal` values in PySpark take two parameters: a **precision** and a **scale**. Precision will be the number of digits total (both integer and decimal part) we want to represent, where scale will be the number of digits in the decimal part. For instance, a `Decimal(10, 2)` will allow eight digits before the decimal point and two after. Taking the same example as we did with floats and double, but applying it to decimals, lead to . I've selected a precision of 10 and a scale of 8, which means 2 digits before the decimal point and 8 after, for a total of 10.

Listing 6.6 Demonstrating the precision of decimal. We don't have loss of precision after performing arithmetic.

```
doubles = spark.createDataFrame([[0.1, 0.2]], ["zero_one", "zero_two"])
decimals = doubles.select(
    [F.col(i).cast(T.DecimalType(10, 8)) for i in doubles.columns]
)
decimals.withColumn(
    "zero_three", F.col("zero_one") + F.col("zero_two") - F.col("zero_one")
).show()

# +----------+----------+----------+
# |  zero_one|  zero_two|zero_three|
# +----------+----------+----------+
# |0.10000000|0.20000000|0.20000000|
# +----------+----------+----------+
```

6.2.4 Date and timestamp

Representing date and time in a format both humans and computers can understand is one of those things that look simple, but really aren't. When you start talking about leap years, leap seconds (yes, they are a thing), timezones and week numbers, it's easy to start making mistakes. On top of this, not everyone in the world agrees about how you should *textually* represent a date, so parsing them can become quite the problem (is 7/11 July 11th or November 7th?).

PySpark provides a type to represent a date without time— `DateType()`— and one with the time included— `TimestampType()`. The timestamp one is much more useful since it is a super-set of the date. (You can think of a date as a timestamp with the time internally set to midnight.) You can naturally parse `datetime.date` and `datetime.datetime` Python objects into PySpark `Date` and `Timestamp` naturally, just like we did for numerical values.

Internally, PySpark will store a timestamp as the number of seconds since the Unix epoch, which is 1970-01-01 00:00:00 UTC (Coordinated Universal Time). The code in demonstrates how an integer is converted to a timezone-aware timestamp object. PySpark will read the integer value, interpret it as the number of seconds since the Unix epoch, and then convert the result into the appropriate timestamp for the timezone your cluster is in. We force a timezone via `spark.conf.set`, which allow the override of the configuration of certain settings of your Spark cluster for the duration of your session. `spark.sql.session.timeZone` contains the timezone information, which is used for displaying timestamp values in an easy to digest format.

Listing 6.7 Converting an integer into a timestamp object according to three timezones

```
integer_values = spark.createDataFrame(
    [[0], [1024], [2 ** 17 + 14]], ["timestamp_as_long"]
)

for ts in ["UTC", "America/Toronto", "Europe/Warsaw"]:
    spark.conf.set("spark.sql.session.timeZone", ts)    ❶
    ts_values = integer_values.withColumn(
        "ts_{}".format(ts), F.col("timestamp_as_long").cast(T.TimestampType())
    )
    print(
        "==={}===".format(spark.conf.get("spark.sql.session.timeZone"))
    )    ❷
    ts_values.show()

# ===UTC===
# +----------------+-------------------+
# |timestamp_as_long|            ts_UTC |
# +----------------+-------------------+
# |               0|1970-01-01 00:00:00|   ❸
# |            1024|1970-01-01 00:17:04|
# |          131086|1970-01-02 12:24:46|
# +----------------+-------------------+
#
# ===America/Toronto===
# +----------------+-------------------+
# |timestamp_as_long| ts_America/Toronto|
# +----------------+-------------------+
# |               0|1969-12-31 19:00:00|   ❹
# |            1024|1969-12-31 19:17:04|
# |          131086|1970-01-02 07:24:46|
# +----------------+-------------------+
#
# ===Europe/Warsaw===
# +----------------+-------------------+
# |timestamp_as_long|   ts_Europe/Warsaw|
# +----------------+-------------------+
# |               0|1970-01-01 01:00:00|   ❺
# |            1024|1970-01-01 01:17:04|
# |          131086|1970-01-02 13:24:46|
# +----------------+-------------------+
```

❶ Spark will infer your timezone based on the timezone settings of your cluster. Here, we force the timezone to be set to a specific one, to demonstrate how the same integer value will yield different timestamp results.

❷ Confirming our timezone via `spark.conf.get`

❸ 0 seconds since `1970-01-01 00:00:00` UTC, converted to UTC, is `1970-01-01 00:00:00`.

❹ Since Toronto is 5 hours behind UTC when daylight saving time isn't in force, we have `1969-12-31 19:00:00` or 5 hours before the Unix epoch.

❺ Warsaw is 1 hour ahead of UTC when daylight saving time isn't in force so we have `1970-01-01 01:00:00`

WARNING Although Pyspark's `Date` objects are similarly constructed, you can't cast an integer into a `Date` object. Doing so will lead to a runtime error.

Most of the time, we won't provide date and timestamp values as an integer: they'll be displayed textually, and PySpark will need to parse them into something useful. Converting date and time from their string representation to a representation you can perform arithmetic and apply functions on is tremendously useful.

SIDEBAR **ISO8601 your way to sanity**

By default, PySpark will display the date and timestamp using something akin to the ISO-8601 format. This means `yyyy-MM-dd HH:mm:ss`. If you have any control over how your data is stored and used, I cannot recommend following the standard enough. It will put to rest the `MM/dd/yyyy` (American format) vs. `dd/MM/yyyy` (elsewhere in the world) ambiguity that drove me nuts when I was growing up. It also solves the AM/PM debate: use the 24-hour system!

If you are curious about the different ways ISO-8601 allows dates to be printed, the Wikipedia page (https://en.wikipedia.org/wiki/ISO_8601) is a pretty good reference. The ISO standard goes more in details, but you'll need to pay for it (https://www.iso.org/iso-8601-date-and-time-format.html).

Two methods are available for reading string values into a date (`to_date()`) and a timestamp (`to_timestamp()`). They both work the same ways.

1. The first, mandatory, parameter is the column you with to convert.
2. The second, optional, parameter is the semantic format of your string column.

PySpark uses Java's `SimpleDateFormat` grammar to parse date and timestamps string values, and also to echo them into string. The full documentation is available on Java's website (https://docs.oracle.com/javase/8/docs/api/java/text/SimpleDateFormat.html) and I recommend you have a read if you plan on parsing dates and timestamps. I use a subset of the functionality available to parse "american-style" date and time.

In , we see that PySpark will naturally read ISO-8601-like string columns. When working with another format, the second parameter (called `format`) provides the simple grammar for reading. shows the format string that will parse `04/01/2019 5:39PM`.

Listing 6.8 Reading string column

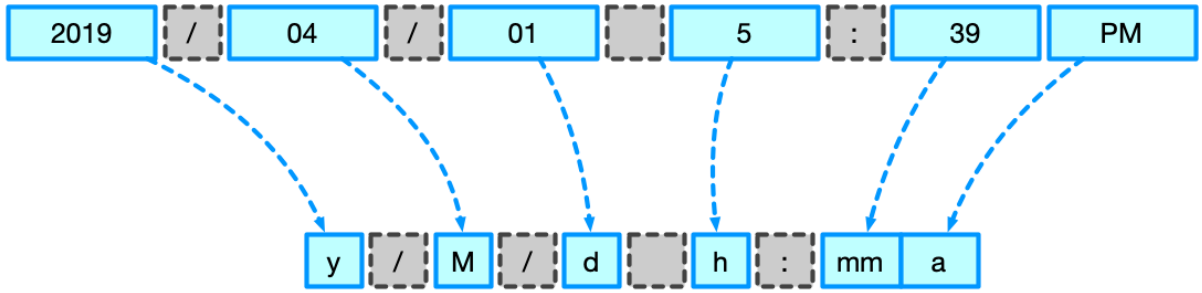
```
spark.conf.set("spark.sql.session.timeZone", "UTC")    ❶

some_timestamps = spark.createDataFrame(
    [["2019-04-01 17:39"], ["2020-07-17 12:45"], ["1994-12-03 00:45"]],
    ["as_string"],
)

some_timestamps = some_timestamps.withColumn(
    "as_timestamp", F.to_timestamp(F.col("as_string"))    ❷
)

some_timestamps.show()

# +----------------+-------------------+
# |       as_string|       as_timestamp|
# +----------------+-------------------+
# |2019-04-01 17:39|2019-04-01 17:39:00|
# |2020-07-17 12:45|2020-07-17 12:45:00|
# |1994-12-03 00:45|1994-12-03 00:45:00|
# +----------------+-------------------+

more_timestamps = spark.createDataFrame(
    [["04/01/2019 5:39PM"], ["07/17/2020 12:45PM"], ["12/03/1994 12:45AM"]],
    ["as_string"],
)

more_timestamps = more_timestamps.withColumn(
    "as_timestamp", F.to_timestamp(F.col("as_string"), "M/d/y h:mma")    ❸
)

more_timestamps.show()

# +------------------+-------------------+
# |         as_string|       as_timestamp|
# +------------------+-------------------+
# | 04/01/2019 5:39PM|2019-04-01 17:39:00|
# |07/17/2020 12:45PM|2020-07-17 12:45:00|
# |12/03/1994 12:45AM|1994-12-03 00:45:00|
# +------------------+-------------------+

this_will_fail_to_parse = more_timestamps.withColumn(
    "as_timestamp", F.to_timestamp(F.col("as_string"))    ❹
)

this_will_fail_to_parse.show()

# +------------------+------------+
# |         as_string|as_timestamp|
# +------------------+------------+
# | 04/01/2019 5:39PM|        null|
# |07/17/2020 12:45PM|        null|
# |12/03/1994 12:45AM|        null|
# +------------------+------------+
```

❶ I am forcing the timezone to UTC so your code will be the same wherever you are on the planet.

❷ The `to_timestamp()` will try to infer the semantic format of the string column. It works well with `yyyy-MM-dd hh:mm` formatted datetime.

❸ You can also pass date formatted differently, as long as you provide a `format` parameter to show PySpark how your values are meant to be read.

❹ If you don't, PySpark will silently default them to `null` values.



Variable portion (what we match against)

Fixed portion (delimiters/formatting)

token	definition
y	Year
M	month
d	day
h	hours
m	minutes
a	AM/PM

Figure 6.3 Parsing 04/01/2019 5:39PM as a timestamp object using PySpark/Java's `SimpleDateFormat's grammar

Once you have your values properly parsed, PySpark provide many functions on date and timestamp columns. `year()`, `month()`, `day()`, `hour()`, `minute()`, `second()` will return the relevant portion of your `Date` or `Timestamp` object, when appropriate. You can add and subtract days using `date_add()` and `date_diff()` and even computer the number of days between two dates using `datediff()` (careful about not adding an underscore!).

Listing 6.9 Manipulating time-stamps with PySpark functions

```
import datetime as d

some_timestamps = (
    spark.createDataFrame(
        [["2019-04-01 17:39"], ["2020-07-17 12:45"], ["1994-12-03 00:45"]],
        ["as_string"],
    )
    .withColumn("as_timestamp", F.to_timestamp(F.col("as_string")))
    .drop("as_string")
)

some_timestamps = (
    some_timestamps.withColumn(
        "90_days_turnover", F.date_add(F.col("as_timestamp"), 90)
    )
    .withColumn("90_days_turnunder", F.date_sub(F.col("as_timestamp"), 90))
    .withColumn(
        "how_far_ahead",
        F.datediff(F.lit(d.datetime(2020, 1, 1)), F.col("as_timestamp")),
    )
    .withColumn(
        "in_the_future",
        F.when(F.col("how_far_ahead") < 0, True).otherwise(False),
    )
)

some_timestamps.show()

# +-------------------+---------------+----------------+-------------+-------------+
# |       as_timestamp|90_days_turnover|90_days_turnunder|how_far_ahead|in_the_future|
# +-------------------+---------------+----------------+-------------+-------------+
# |2019-04-01 17:39:00|     2019-06-30|      2019-01-01|          275|        false|
# |2020-07-17 12:45:00|     2020-10-15|      2020-04-18|         -198|         true|
# |1994-12-03 00:45:00|     1995-03-03|      1994-09-04|         9160|        false|
# +-------------------+---------------+----------------+-------------+-------------+
```

Finally, date and timestamp columns are incredibly useful when doing time-series analysis and when using windowing. We will cover this in Chapter 9.

6.2.5 Null and boolean

Those types are not related, but they are so simple in their construction that they don't warrant a section to themselves.

A boolean in PySpark is simply a bit (meaning a `0` or `1` value), where 0 means false and 1 means true. It is by definition the smallest bit of information a piece of data can carry (1 bit can encode 2 values, here `True/False`). It is just like booleans in most other programming languages, including Python, with one key exception. In Python, every type evaluate implicitly to a Boolean, which allows code like such.

```
trois = 3

if trois:
   print("Trois!)

# Trois!
```

PySpark's conditionals, available through the `when()` function (Chapter 2), will require a Boolean test. You can't just use `F.when(F.col("my_column"), "value_if_true")` as this will result in a runtime error. One advantage of requiring explicit booleans for truth testing is that you don't have to remember what equates to `True` and what equates to `False`.

Null values in PySpark represent the absence of value. It is most similar to `None` in Python. Just like in SQL (which we'll cover briefly in Chapter 7), null values are absorbing in PySpark, meaning that passing null as an input to a Spark function will return a null value. This is standard behavior in most data processing libraries. When creating our custom functions, called UDF or *User Defined Functions*, gracefully dealing with null values will be important to keep that consistency.

`null` in PySpark is consistent with how SQL treats them, which means it's a little different than other data manipulations libraries, such as Pandas. In PySpark and Spark, `null` is a value just like any other one, and will be kept when applying functions. The best canonical example is when you `groupBy` a data frame containing `null` values in the key. Where would delete those null keys by default, PySpark will keep them. In general, I prefer this consistent approach of considering all `null` to be a distinct value, as it avoids dropping columns. It makes exploratory data analysis a whole lot easier since null values are displayed when summarizing data. On the other hand, it means that filtering for nulls— if you want to get rid of them— is something you need to remember explicitly.

In PySpark, testing for null values within a column is done through the `isNull()` and `isNotNull()` methods (careful about the case!).

Listing 6.10 Grouping a data frame containing null values

```
some_nulls = spark.createDataFrame(
    [[1], [2], [3], [4], [None], [6]], ["values"]    ❶
)

some_nulls.groupBy("values").count().show()

# +------+-----+
# |values|count|
# +------+-----+
# |  null|    1|    ❷
# |     6|    1|
# |     1|    1|
# |     3|    1|
# |     2|    1|
# |     4|    1|
# +------+-----+

some_nulls.where(F.col("values").isNotNull()).groupBy(
    "values"
).count().show()    ❸

# +------+-----+
# |values|count|
# +------+-----+
# |     6|    1|
# |     1|    1|
# |     3|    1|
# |     2|    1|
# |     4|    1|
# +------+-----+
```

❶ One of our values is null (converted from Python's `None`)

❷ Grouping our column creates a grouping for the `null` values

❸ Removing the `null` values with a `where` method.

Null values can create problems when creating a data frame. PySpark will not be able to infer the type of a column if it only contains only null values. The simplest example is provided in , but this can also happen when reading CSV files where there aren't any values in one of the columns. When confronted with this problem, you can provide your own schema information. This is covered in .

Listing 6.11 Reading a null column leads to a `TypeError`

```
this_wont_work = spark.createDataFrame([None], "null_column")

# A heap of stack traces.
#
# [...]
#
# TypeError: Can not infer schema for type: <class 'NoneType'>
```

6.3 PySpark's complex types

PySpark's ability to use complex types inside the data frame is what allows its remarkable flexibility. While you still have the tabular abstraction to work with, your cells are supercharged since they can contain more than just a value. It's just like going from 2D to 3D, and even beyond!

A *complex* type isn't really complex: I often use the term *container* or *compound* type and will interchangeably during the book. In a nutshell, the difference between them and simple/scalar types is their ability to contain more than a single value. In Python, the main complex types are the list, the tuple, and the dictionary. In PySpark, we have the array, the map, and the struct. With those 3, you will be able to express an infinite amount of data layout.

6.3.1 Complex types: the array

The simplest complex type in PySpark is the array. It is not our first encounter: we used it naturally in Chapter 2 when we transformed lines of text into arrays of words.

A PySpark array can be loosely thought as a Python list or a tuple. It is a container for multiple unordered elements **of the same type**. When using the type constructor for an array, you need to specify the type of elements it will contain. For instance, an array of longs will be written as a `ArrayType(LongType())`. shows a simple example of an array column inside a data frame. You can see that the display syntax is identical to the Python one for lists, using the square bracket and a comma as a delimiter. Because of the default behavior of `show()` in PySpark, it's always better to remove the truncation to see a little bit more inside the cell.

Listing 6.12 Making a PySpark data frame containing an array column, using Python lists

```
array_df = spark.createDataFrame(
    [[[1, 2, 3]], [[4, 5, 6]], [[7, 8, 9]], [[10, None, 12]]], ["array_of_ints"]
)

array_df.printSchema()

# root
# |-- array_of_ints: array (nullable = true)   ❶
# |    |-- element: long (containsNull = true)   ❷

array_df.show()

# +-------------+
# |array_of_ints|
# +-------------+
# |    [1, 2, 3]|
# |    [4, 5, 6]|
# |    [7, 8, 9]|
# |    [10,, 12]|   ❸
# +-------------+
```

❶ The type of the column we created is `array`

❷ Every `array` column contains a sub-entry called `element`. Here, our elements are `long`, to the resulting type is `Array(Long)`

❸ An array can contain null values. Of course, reading an column with only arrays filled with null values will yield a `TypeError`, just like for columns

In Chapter 2, we saw a very useful method— `explode()`— for converting arrays in rows of elements. PySpark provides many other methods to work with arrays. This time, the API is quite friendly: the methods are prefixed with `array_*`, when working with a single array column, or `arrays_*` when working with two or more. As an example, let's take an extended version of rock-paper-scissors: the Pokemon type chart. Each Pokemon can have one or more types, which makes perfect sense for an array. The file we'll be reading our information from comes from Bulbapedia[7], which I've cleaned to keep the canonical type for each Pokemon. The file is a DSV using the tab character as a delimiter. We'll use the array type to answer a quick question: which is the most popular *dual-type* (Pokemons with two types) in the Pokedex?

The code in exposes one way to answer the question. We read the DSV file, and then take the first and second type into an array. Since single-type Pokemons have a second type of `null`, which is cumbersome to explicitly remove, we duplicate their type before building the array (for example, `[Fire,]  [Fire, Fire]`). Once the arrays are built, we dedupe the types, keep only the dual ones (where the type array contains more than a single deduped value). It's then just a matter of group, count, order, and print, which we've done plenty.

Listing 6.13 Identifying the most popular dual-type Pokemons with PySpark's arrays

```
pokedex = spark.read.csv("../../data/Ch06/pokedex.dsv", sep="\t").toDF(
    "number", "name", "name2", "type1", "type2"    ❶
)

pokedex.show(5)

# +------+----------+----------+-----+------+
# |number|      name|     name2|type1| type2|
# +------+----------+----------+-----+------+
# |  #001| Bulbasaur| Bulbasaur|Grass|Poison|
# |  #002|   Ivysaur|   Ivysaur|Grass|Poison|
# |  #003|  Venusaur|  Venusaur|Grass|Poison|
# |  #004|Charmander|Charmander| Fire|  null|
# |  #005|Charmeleon|Charmeleon| Fire|  null|
# +------+----------+----------+-----+------+
# only showing top 5 rows

pokedex = (
    pokedex.withColumn(
        "type2",
        F.when(F.isnull(F.col("type2")), F.col("type1")).otherwise(    ❷
            F.col("type2")
        ),
    )
    .withColumn("type", F.array(F.col("type1"), F.col("type2")))    ❸
    .select("number", "name", "type")
)

pokedex.show(5)

# +------+----------+---------------+
# |number|      name|           type|
# +------+----------+---------------+
# |  #001| Bulbasaur|[Grass, Poison]|
# |  #002|   Ivysaur|[Grass, Poison]|
# |  #003|  Venusaur|[Grass, Poison]|
# |  #004|Charmander|   [Fire, Fire]|
# |  #005|Charmeleon|   [Fire, Fire]|
# +------+----------+---------------+
# only showing top 5 rows

pokedex.withColumn("type", F.array_sort(F.col("type"))).withColumn(
    "type", F.array_distinct(F.col("type"))    ❹
).where(
    F.size(F.col("type")) > 1    ❺
).groupby(
    "type"
).count().orderBy(
    "count", ascending=False
).show(
    10
)

# +-----------------+-----+
# |             type|count|
# +-----------------+-----+
# | [Flying, Normal]|   27|
# |  [Grass, Poison]|   14|
# |    [Bug, Flying]|   13|
# |    [Bug, Poison]|   12|
# |    [Rock, Water]|   11|
# |  [Ground, Water]|   10|
# |   [Ground, Rock]|    9|
# |  [Flying, Water]|    8|
```

```
# | [Fairy, Psychic]|    7|
# |[Flying, Psychic]|    7|
# +-----------------+-----+
# only showing top 10 rows
```

❶ We use the `toDF()` method to specify our column names since the file doesn't have a header

❷ Since the array can contain null values, we duplicate the first type if the second one is null. Since `null` doesn't equal to himself, we avoid the problem of removing null values from the array (see question X.X) //TODO!

❸ We build our array using the `array()` function, which will clump columns of the same type in a single array column.

❹ We remove duplicate types (single-type Pokemons) with the `array_distinct()` function.

❺ We keep only dual-type Pokemons using the `size` function, which will compute the length of any compound structure inside a column.

Finally, one neat aspect of the array is that you can access elements by position just like a Python list, using the bracket notation. Should we wanted to keep only the first type for each Pokemon once they're clumped in an array, we can simply use `[index]` on the column. Should you ask for an index beyond what the array contains, PySpark will return null for the rows.

```
pokedex.select(F.col("type")[0])
```

6.3.2 Complex types: the map

Where we can approximate an array to a list, a PySpark map is akin to a Python dictionary. Unlike Python dictionaries and just like PySpark arrays, a map needs to have consistent types for keys and values. A map containing string keys and integer values will have the type `MapType(StringType(), IntegerType())`, as an example. shows a simple example about a map column inside a data frame, using this time a simpler rock, paper, scissor example.

Creating a map from existing columns is done through the `create_map()` function. This function takes an even number of arguments and will assign the odd columns passed as arguments to the map keys, and the even as map values. Just like an array, the bracket syntax works, but this time we need to pass the key as the argument. If the key doesn't exist, `null` will be returned. You can also use the dot notation: in our case, the field after the dot would either be `key` or `value`.

A few functions for the map are available, and just like for arrays, they are prefixed by `map_*`.

Listing 6.14 Using a map to encode the rules of rock, paper, scissor

```
# Choice, strong against, weak against
rock_paper_scissor = [
    ["rock", "scissor", "paper"],
    ["paper", "rock", "scissor"],
    ["scissor", "paper", "rock"],
]

rps_df = spark.createDataFrame(
    rock_paper_scissor, ["choice", "strong_against", "weak_against"]
)

rps_df = rps_df.withColumn(
    "result",
    F.create_map(
        F.col("strong_against"),
        F.lit("win"),
        F.col("weak_against"),
        F.lit("lose"),    ❶
    ),
).select("choice", "result")

rps_df.printSchema()

# root
# |-- choice: string (nullable = true)
# |-- result: map (nullable = false)   ❷
# |    |-- key: string
# |    |-- value: string (valueContainsNull = false)

rps_df.show(3, False)

# +-------+------------------------------+
# |choice |result                        |
# +-------+------------------------------+
# |rock   |[scissor -> win, paper -> lose]|   ❸
# |paper  |[rock -> win, scissor -> lose] |
# |scissor|[paper -> win, rock -> lose]   |
# +-------+------------------------------+

rps_df.select(
    F.col("choice"), F.col("result")["rock"], F.col("result.rock")    ❹
).show()

# +-------+------------+----+
# | choice|result[rock]|rock|
# +-------+------------+----+
# |   rock|        null|null|
# |  paper|         win| win|
# |scissor|        lose|lose|
# +-------+------------+----+

rps_df.select(F.map_values(F.col("result"))).show()    ❺

# +------------------+
# |map_values(result)|
# +------------------+
# |       [win, lose]|
# |       [win, lose]|
# |       [win, lose]|
# +------------------+
```

❶ The `strong_against` column will map to the literal value "win" and the `weak_against` column will map to "lose".

❷ The `result` column here is a `map` containing two elements, a `key` and `value`

❸ PySpark displays maps like an array, but where each key maps to a value with an arrow .

❹ The bracket syntax, passing the key as a parameter, works as expected, as is the dot notation.

❺ In order to extract only the values as an array, the function `map_values()` is used.

6.4 Structure and type: The dual-nature of the struct

One of the key weaknesses of the map or the array is that you can't represent heterogeneous collections. All of your elements, key or values need to be of the same type. The struct is different: it will allow for elements to be of a different type.

Fundamentally, a struct is like a map/dictionary, but with string keys and typed value. Each block of the struct is defined as a `StructField`. Should we want to represent a struct containing a string, a long and a timestamp value, our struct would be defined as such.

```
T.StructType(
  [
    T.StructField("string_value", T.StringType()),
    T.StructField("long_value", T.LongType()),
    T.StructField("timestamp_value", T.TimestampType())
  ]
)
```

If you squint a little, you might think that this looks like a data frame schema, and you'd be right. As a matter of fact, **PySpark encodes rows as structs**, or if you prefer, **the struct is the building block of a PySpark data frame**. Since you can have a struct as a column, you can nest data frames into one another and represent hierarchical or complex data relationships.

I will take back our Pokedex dataset from to illustrate the dual nature of the struct. For the first time, I will explicitly define the schema and pass it to the data frame ingestion.

The `StructType` takes a list of `StructField`. Those `StructField` take two mandatory parameters and two optional.

1. The first one is the name of the field, as a string
2. The second one is the type of the field as a PySpark type
3. The third one is a boolean flag that lets PySpark know if we expect null values to be present.
4. A dictionary containing string keys and simple values for metadata (mostly used for ML pipelines, Chapter 13)

Once your data frame is created, the way to add a column is to use the `struct()` column, from `pyspark.sql.functions`. It will create a struct containing the columns you pass as an

argument. In , I use the two purposes of the struct. First, I create a schema that matches my data in DSV format, using `StructType` and a list of `StructField` to define the names of the columns and their types. Once the data loaded in a data frame, I create a struct column, using the `name`, `type1` and `type2` field. Displaying the resulting data frame's schema shows that the struct adds a second level of hierarchy to our schema, just like the array and the map. This time, we have full control over the name, types, and number of fields. When requested to `show()` a sample of the data, PySpark will simply wrap the fields of the struct into square brackets and make it look like an array of values.

Listing 6.15 Using the two natures of the struct (container and type)

```
pokedex_schema = T.StructType(    ❶
    [
        T.StructField("number", T.StringType(), False, None),
        T.StructField("name", T.StringType(), False, None),
        T.StructField("name2", T.StringType(), False, None),
        T.StructField("type1", T.StringType(), False, None),
        T.StructField("type2", T.StringType(), False, None),
    ]
)

pokedex = spark.read.csv(
    "../../data/Ch06/pokedex.dsv", sep="\t", schema=pokedex_schema
)

pokedex.printSchema()

# root
# |-- number: string (nullable = true)
# |-- name: string (nullable = true)
# |-- name2: string (nullable = true)
# |-- type1: string (nullable = true)
# |-- type2: string (nullable = true)

pokedex_struct = pokedex.withColumn(
    "pokemon", F.struct(F.col("name"), F.col("type1"), F.col("type2"))    ❷
)

pokedex_struct.printSchema()

# root
# |-- number: string (nullable = true)
# |-- name: string (nullable = true)
# |-- name2: string (nullable = true)
# |-- type1: string (nullable = true)
# |-- type2: string (nullable = true)
# |-- pokemon: struct (nullable = false)   ❸
# |    |-- name: string (nullable = true)
# |    |-- type1: string (nullable = true)
# |    |-- type2: string (nullable = true)

pokedex_struct.show(5, False)

# +------+----------+----------+-----+------+-------------------------+
# |number|name      |name2     |type1|type2 |pokemon                  |
# +------+----------+----------+-----+------+-------------------------+
# |#001  |Bulbasaur |Bulbasaur |Grass|Poison|[Bulbasaur, Grass, Poison]|   ❹
# |#002  |Ivysaur   |Ivysaur   |Grass|Poison|[Ivysaur, Grass, Poison]  |
# |#003  |Venusaur  |Venusaur  |Grass|Poison|[Venusaur, Grass, Poison] |
# |#004  |Charmander|Charmander|Fire |null  |[Charmander, Fire,]       |
# |#005  |Charmeleon|Charmeleon|Fire |null  |[Charmeleon, Fire,]       |
# +------+----------+----------+-----+------+-------------------------+

pokedex_struct.select("name", "pokemon.name").show(5)    ❺

# +----------+----------+
# |      name|      name|
# +----------+----------+
# | Bulbasaur| Bulbasaur|
# |   Ivysaur|   Ivysaur|
# |  Venusaur|  Venusaur|
# |Charmander|Charmander|
# |Charmeleon|Charmeleon|
# +----------+----------+

pokedex_struct.select("pokemon.*").printSchema()    ❻
```

```
# root
# |-- name: string (nullable = true)
# |-- type1: string (nullable = true)
# |-- type2: string (nullable = true)
```

❶ I build a manual data frame schema through `StructType()`, which takes a list of `StructFields()`.

❷ We can create a struct column with the `struct()` function. I am passing 3 columns as parameters, so my struct has those 3 fields.

❸ A `struct` adds another dimension into the data frame, containing an arbitrary number of fields, each having their types.

❹ A struct is displayed just like an array of the value its fields contain

❺ I use the dot notation to extract the `name` field inside the `pokemon` struct

❻ To "flatten" the struct, we can use the `.*` notation. The star refers to "every field in the struct"

To extract the value from a struct, we can use the dot or bracket notation just like we did on the map. If you want to select all fields in the struct, PySpark provides a special star field to do so. `pokemon.*` will give you all the fields of the `pokemon` struct.

6.4.1 A data frame is an ordered collection of columns

When working with a data frame, we are really working with two structures:

1. The data frame itself, via methods like `select()`, `where()` or `groupBy()` (which, to be pedantic, gives you a `GroupedData` object, but we'll consider them to be analogous)
2. The `Column` object, via functions (from `pyspark.sql.functions` and UDF (Chapter 8)) and methods such as `cast()` (see) and `alias()`)

The data frame keeps the order of your data by keeping an ordered collection of columns. Column name and type are managed at runtime, just like regular Python. It is your responsibility as the developer to make sure you're applying the right function to the right structure, and that your types are compatible. Thankfully, PySpark makes it easy and obvious to peek at the data frame's structure, via `printSchema()` or simply inputting your data frame name on the REPL.

SIDEBAR **A statically typed option: the data set**

Python, Java, and Scala are all strongly typed languages, meaning that performing an illegal operation will raise an error. Java and Scala are statically typed on top of that, meaning that the types are known at compilation time. Type Errors will lead to a compilation error in Java/Scala, where Python will raise a run-time error. The same behavior will happen using the data frame since the types are dynamic.

Spark with Scala or Java also provides a statically typed version of the data frame, called the data set (or `DataSet[T]`, where `T` are the types inside the data set). When working with a data set in Spark, you know at compile time the types of your columns. For instance, our data frame in would have been a `DataSet[String, Integer, Double, Timestamp]`, and doing an illegal operation (such as adding 1 to a string) would yield a compile-time error.

The `DataSet` structure is not available for PySpark or Spark for R, since those languages aren't statically typed. It wouldn't make much sense to forego the nature of Python, even when using Spark. Just like when coding with Python, we'll have to be mindful of our types. When working with integrating Python functions as User Defined Functions (mostly in Chapter 8), we'll use mypy (http://mypy-lang.org/), an optional static type checker for Python. It will help a little by avoiding easy type errors.

PySpark putting so much emphasis on columns is not something new nor revolutionary. SQL databases have been doing this for decades now. In fact, Chapter 7 is all about using SQL within PySpark! Unlike SQL, PySpark treats the column like a bona-fide object, which means you can Python your way into where you want to go. When I was myself learning PySpark, this stumped me for quite some time. I believe an example demonstrates it best.

Since PySpark exposes `Column` as an object, we can apply functions on it or use its methods. This is what we've been doing since Chapter 2. The neat aspect of separating columns from its parent structure (the data frame) is that you can code both separately. PySpark will keep the chain of transformations of the column, *independently from the data frame you want to apply it to*, and wait for a transformation, just like any other data frame transformation. In practice, it means that you can simplify or encapsulate your transformations. A simple example is demonstrated in . We create a variable `transformations` which is a list of two transformations, a column rename, and a `when` transformation. When defining `transformations`, PySpark knows nothing about the data frame it'll be applied two, but happily stores the chain of transformations, trusting that it'll be applied to a compatible data frame later.

Listing 6.16 Simplifying our transformations using Python and the `Column` object

```
pokedex = spark.read.csv("../../data/Ch06/pokedex.dsv", sep="\t").toDF(
    "number", "name", "name2", "type1", "type2"    ❶
)

transformations = [
    F.col("name").alias("pokemon_name"),
    F.when(F.col("type1").isin(["Fire", "Water", "Grass"]), True)
    .otherwise(False)
    .alias("starter_type"),
]    ❶

transformations

# [Column<b'name AS `pokemon_name`'>,
# Column<b'CASE WHEN (type1 IN (Fire, Water, Grass)) THEN true ELSE false END AS `starter_type`'>]

pokedex.select(transformations).printSchema()    ❷

# root
# |-- pokemon_name: string (nullable = true)
# |-- starter_type: boolean (nullable = false)

pokedex.select(transformations).show(5, False)

# +------------+------------+
# |pokemon_name|starter_type|
# +------------+------------+
# |Bulbasaur   |true        |
# |Ivysaur     |true        |
# |Venusaur    |true        |
# |Charmander  |true        |
# |Charmeleon  |true        |
# +------------+------------+
```

❶ We can store our column transformations into a variable (here a Python list called `transformations`).

❷ I apply the chain of transformations to a data frame. It works like if I wrote the list inside the select itself.

6.4.2 The second dimension: just enough about the row

Although a PySpark data frame is all about the columns, there is a Row object we can access. A row is very similar to a Python dictionary: you'll have a key-value pair for each column. You can in fact use the `asDict()` method for marshalling the `Row` into a python dict.

The biggest usage of the `Row` object is when you want to operate the data frame as an RDD. As we will see in Chapter 8, an RDD is closer to row-major as a structure, rather than the column-major nature of the data frame. When converting a data frame into an RDD, PySpark will create an RDD of `Rows`. The same applies: if you have an RDD of `Rows`, it's very easy to convert it into a data frame.

Listing 6.17 Listing title

```
row_sample = [
    T.Row(name="Bulbasaur", number=1, type=["Grass", "Poison"]),    ❶
    T.Row(name="Charmander", number=4, type=["Fire"]),
    T.Row(name="Squirtle", number=7, type=["Water"]),
]

row_df = spark.createDataFrame(row_sample)    ❷

row_df.printSchema()

# root
# |-- name: string (nullable = true)
# |-- number: long (nullable = true)
# |-- type: array (nullable = true)
# |    |-- element: string (containsNull = true)

row_df.show(3, False)

# +----------+------+---------------+
# |name      |number|type           |
# +----------+------+---------------+
# |Bulbasaur |1     |[Grass, Poison]|
# |Charmander|4     |[Fire]         |
# |Squirtle  |7     |[Water]        |
# +----------+------+---------------+

row_df.take(3)

# [Row(name='Bulbasaur', number=1, type=['Grass', 'Poison']),
#  Row(name='Charmander', number=4, type=['Fire']),
#  Row(name='Squirtle', number=7, type=['Water'])]
```

❶ The `Row` object is constructed with an arbitrary number of fields, which do not need to be quoted.

❷ PySpark will accept a list of `Row` to `createDataFrame` directly

You can also `take` a number `n` of rows into a Python list of `Rows` to manipulate them further locally. In practice, you'll see that the conversion from PySpark data frames to a local pandas data frame to be much more common. I'll cover this in greater detail in Chapter 9. You'll seldom use the `Row` object directly when working with structured data but it's nice to know it's there and ready for you should you need the added flexibility.

WARNING `take()` always takes a parameter `n` specifying the number of rows you want. Careful not to pass a too large number since it'll create a local Python list on the master node.

6.4.3 Casting your way to sanity

Now armed with knowledge about types, we can now look at changing the type of columns in PySpark. This is a common but dangerous operation, as we can either crash our program or silently default values to null, as we saw in .

The casting of a column is via a method on the column itself, called `cast()` or `asType()` (both are synonyms). It is usually used in conjunction with a method on the data frame, most often `select()` or `withColumn()`. The code in shows the most common types of casting: from and to a string field.

The `cast()` method is very simple. Applied to a `Column` object, it takes a single parameter, which is the type you want the column to become. If PySpark can't make the conversion, it'll nullify the value.

Listing 6.18 Casting values using the `cast()` function

```
data = [
    ["1.0", "2020-04-07", "3"],
    ["1042,5", "2015-06-19", "17,042,174"],
    ["17.03.04178", "2019/12/25", "17_092"],
]

schema = T.StructType(
    [
        T.StructField("number_with_decimal", T.StringType()),
        T.StructField("dates_inconsistently_formatted", T.StringType()),
        T.StructField("integer_with_separators", T.StringType()),
    ]
)

cast_df = spark.createDataFrame(data, schema)

cast_df.show(3, False)
# +------------------+-----------------------------+----------------------+
# |number_with_decimal|dates_inconsistently_formatted|integer_with_separators|
# +------------------+-----------------------------+----------------------+
# |1.0               |2020-04-07                   |3                     |
# |1042,5            |2015-06-19                   |17,042,174            |
# |17.03.04178       |2019/12/25                   |17_092                |
# +------------------+-----------------------------+----------------------+

cast_df = cast_df.select(
    F.col("number_with_decimal")
    .cast(T.DoubleType())    ❶
    .alias("number_with_decimal"),
    F.col("dates_inconsistently_formatted")
    .cast(T.DateType())    ❷
    .alias("dates_inconsistently_formatted"),
    F.col("integer_with_separators")
    .cast(T.LongType())    ❸
    .alias("integer_with_separators"),
)

cast_df.show(3, False)

# +------------------+-----------------------------+----------------------+
# |number_with_decimal|dates_inconsistently_formatted|integer_with_separators|
# +------------------+-----------------------------+----------------------+
# |               1.0|                   2020-04-07|                     3|
# |              null|                   2015-06-19|                  null|
# |              null|                         null|                  null|    ❹
# +------------------+-----------------------------+----------------------+
```

❶ Casting our string-encoded decimal numbers into a double

❷ Casting our string-encoded dates into a date

❸ Casting our string-encoded integers into a long

❹ When PySpark can't convert the value, it silently outputs a `null`.

I consider casting the black-beast of data processing. Conceptually, it shouldn't be hard: we are simply re-encoding values in a format more suitable. In practice, there are so many corner cases. Taking back our original `cast_df` from , PySpark fails to convert five out of nine records we've passed to it, and will silently convert them to null. When working with large data sets, this "silent but deadly" behaviour can mean you're losing data without realizing it later in your processing. There is no surefire way to prevent this completely, but keeping a copy of your previous column and making sure you're not increasing the number of nulls is a good way to make sure you're not having casting problems. `groupBy` or `drop_duplicates()` is also a good way to check quickly for potential problems.

Listing 6.19 Casting carefully, diagnosing the results before signing on them

```
data = [[str(x // 10)] for x in range(1000)]

data[240] = ["240_0"]
data[543] = ["56.24"]
data[917] = ["I'm an outlier!"]    ❶

clean_me = spark.createDataFrame(
    data, T.StructType([T.StructField("values", T.StringType())])
)

clean_me = clean_me.withColumn(
    "values_cleaned", F.col("values").cast(T.IntegerType())
)

clean_me.drop_duplicates().show(10)

# +---------------+--------------+
# |         values|values_cleaned|
# +---------------+--------------+
# |             48|            48|
# |             81|            81|
# |              6|             6|
# |             94|            94|
# |          240_0|          null| <---+
# |I'm an outlier!|          null| <---+--  ❷
# |             41|            41|
# |             18|            18|
# |             47|            47|
# |             89|            89|
# +---------------+--------------+

clean_me = clean_me.withColumn(
    "values", F.split(F.col("values"), "_")[0]
)    ❸
```

❶ I changed three columns to throw a wrench in our casting

❷ Using `drop_duplicates()` reduces the number of records to analyze. `groupby()` can perform the same thing.

❸ As a simple example, we split the strings at the underscore character and keep the group before. This will take c are of `240_0`.

Knowing how to tackle corner cases of casting is very dependent on what your data means. There are no rules on how `56.24` should be encoded as an integer.

1. You could round up (57)
2. You could round down or truncate (56)
3. You could forego the decimal point (5624)

This is a very small foray in the rich and fascinating world of data semantic, or *what is the meaning behind your data?* It sits at the crossroad between experience, domain-knowledge and a little bit of intuition. Remember that data cleaning, which includes casting the right type,

TIP For casting dates and timestamps, since their formatting is much more free-form than numerical values, you can use the `to_date()` and `to_timestamp()` functions. If you need a refresher, head up to .

SIDEBAR **The perils of CSV**

CSV is by far the most popular interchange format for tabular data, and there is no going around it. It's easy to produce and human-readable. That being said, it suffers from three capital flaws for working with PySpark efficiently:

1. it can't represent nested data (maps, arrays, and nested structs);
2. the exporting process needs to be careful with field delimiters (to avoid confusing a comma in a string field versus a comma as a delimiter, for instance);
3. more importantly, everything is a string, so you need to cast everything when you read it.

Those issues are common to other data processing libraries, but since PySpark will usually deal with larger data sets, those flaws are magnified since you can't reasonably inspect every record to make sure everything is done the right way. In Chapter 9, I discuss other file formats that address those issues, at the expense of not being readable with Excel or a text editor.

6.4.4 Defaulting values with fillna

Filling null values in your data frame is the perfect intersection between types and semantic. You need to provide a value that will suit your use-case but also respect the type of the column. You could not, for instance, transform your null values into "NOTHING" if your column is filled with integers.

The method for filling null values is called `fillna()`. This is a method of the data frame (like `where` and `select`). It takes two parameters, and both are extremely important:

1. The first one is the value you wish to replace the null values with. The type of the value will determine which columns it can be applied to.
2. The second is an optional list of columns you want to apply the filling on. If you don't provide one, PySpark will apply the fill to all the compatible type columns.

There are two ways to use the method. The first one is to specify a scalar value (like `-1`, `False` or `"N/A"`) and let PySpark do the application automatically. If you need a more fine-tuned approach, you can pass a dictionary as a first parameter. The keys will map to column names, and the values will be what to replace the null values for that specific column only. In , I demonstrate both use-cases.

`fillna()` will only accept scalar values that are float, int, long, string or bool. Passing another type of value will give a `ValueError`.

Listing 6.20 Filling nulls using the `fillna()` method, using the scalar and dict approach.

```
cast_df.show(3, False)    ❶
# +------------------+----------------------------+----------------------+
# |number_with_decimal|dates_inconsistently_formatted|integer_with_separators|
# +------------------+----------------------------+----------------------+
# |               1.0|                  2020-04-07|                     3|
# |              null|                  2015-06-19|                  null|
# |              null|                        null|                  null|
# +------------------+----------------------------+----------------------+

cast_df.fillna(-1).show()    ❷

# +------------------+----------------------------+----------------------+
# |number_with_decimal|dates_inconsistently_formatted|integer_with_separators|
# +------------------+----------------------------+----------------------+
# |               1.0|                  2020-04-07|                     3|
# |              -1.0|                  2015-06-19|                    -1|
# |              -1.0|                        null|                    -1|
# +------------------+----------------------------+----------------------+

cast_df.fillna(-1, ["number_with_decimal"]).fillna(-3).show()    ❸

# +------------------+----------------------------+----------------------+
# |number_with_decimal|dates_inconsistently_formatted|integer_with_separators|
# +------------------+----------------------------+----------------------+
# |               1.0|                  2020-04-07|                     3|
# |              -1.0|                  2015-06-19|                    -3|
# |              -1.0|                        null|                    -3|
# +------------------+----------------------------+----------------------+

cast_df.fillna({"number_with_decimal": -1, "integer_with_separators": -3}).show()    ❹

# +------------------+----------------------------+----------------------+
# |number_with_decimal|dates_inconsistently_formatted|integer_with_separators|
# +------------------+----------------------------+----------------------+
# |               1.0|                  2020-04-07|                     3|
# |              -1.0|                  2015-06-19|                    -3|
# |              -1.0|                        null|                    -3|
# +------------------+----------------------------+----------------------+
```

❶ I'm using the `cast_df` data frame from .

❷ PySpark will use the same type promotion for arithmetic values we've seen in . In this case, `-1` was applied to both integer and floating-point columns.

❸ You can chain `fillna()` repeatedly. They will be applied sequentially.

❹ The dictionary method can also be used if you need to be precise with how you fill certain columns.

Choosing the right value is dependant on your use-case. For instance, you might want to set all null integers to -1, before realizing this won't work if you use the column for machine learning. Because PySpark treats null values like any other, keeping them around is not harmful as long as you know how to account for them.

This Chapter might have been more academic than necessary. On the other hand, it contained a lot of essential information about how PySpark "thinks" about the values it processes. Knowing

this will avoid a lot of guessing when you build your own data manipulation programs. Remembering how your data needs the right type, the right structure will give you head-start in figuring out how to organize your program so it can be maintainable and resilient.

Don't worry if you weren't able to absorb everything in one read. You can always jump back to the specific section as you need it. I also recommend keeping exploration notes when you're discovering new data. Patterns will emerge and you'll develop an intuition, which ultimately will make you faster. And the faster you'll be at processing data into what you want, the more time you can spend building cool products!

6.5 Summary

- PySpark has a wide set of types that it uses to convey how data is encoded within a given column. Types determine which functions can be applied to a given column.
- There are two main categories of types. The first one is *scalar* or *simple* types, which include string, numbers, both integral and decimal, date and time, boolean, and null values. The second one is *compound* or *complex* types, which are akin to container structures in other programming languages. PySpark provides the array, the map and the struct as compound types.
- The struct is both a column type and how PySpark constructs a data frame. You can build a data frame schema using `StructType`, where each field will be encoded in `StructFields`.
- Casting columns from a type to another is done through the `cast()` or `astype()` method. Casting a value into an incompatible type will yield a null value.
- Null values in PySpark are similar to how they are treated in other SQL databases, where null is another distinct value. You can identify null values within a column using `isNull()/isNotNull()` and replace null values with `fillna()`.

6.6 Exercises

6.6.1 Exercise 6.1

How could you demonstrate that PySpark stores `Date` as `Timestamp` with the time forced to 00:00:00?

6.6.2 Exercise 6.2

Taking the `cast_df` from , how could you fill the null values with the date `1900-01-01`?

This chapter covers

- How PySpark's own data manipulation module takes inspiration from SQL's vocabulary and way of doing things.
- How to register data frames as temporary views or tables to query them using Spark SQL.
- How the catalog stores metadata about registered tables and views, how to list the existing references and delete them.

- How common data manipulations are expressed in PySpark and Spark SQL and how you can move from one to the other.
- How to use SQL-style clauses inside certain PySpark methods.

7

Bilingual PySpark: blending Python and SQL code

My answer to "Python versus SQL, which one should I learn?" is "yes".

When it comes to manipulating tabular data, SQL is the reigning king. For multiple decades now, it has been the workhorse language for relational databases, and even today, learning how to tame it is a worthwhile exercise. Spark acknowledge the power of SQL heads on: you can use a mature SQL API to transform data frames. On top of that, you can also seamlessly blend SQL code withing your Spark or PySpark program, making it easier than ever to migrate those old SQL ETL jobs without reinventing the wheel.

This chapter is dedicated on SQL interop with PySpark. I will cover how we can move from one language to the other. I will also cover how we can use a SQL-like syntax within data frame methods to speed up your code and some of trade-offs you can face. Finally, we'll blend Python and SQL code together to get the best of both worlds.

If you already have notable exposure to SQL, this chapter will be a breeze for you. Feel free to skim over the SQL-specific sections, but don't skip the sections on Python and SQL interop, as there is some PySpark idiosyncrasies I cover there. For those brand new to SQL, this will be— I hope— an eye opener moment and you'll add another tool under your belt. If you feel that you'd like a deeper dive into SQL, *SQL in motion* by Ben Brumm (Manning, 2017) is a good video source. If you prefer a book, a very exhaustive reference is *Joe Celko's SQL for Smarties* (Morgan Kauffman, 2014).

In order to follow the examples in this chapter, here are the imports I am using.

```
from pyspark.sql import SparkSession
from pyspark.sql.utils import AnalysisException    ❶
import pyspark.sql.functions as F
import pyspark.sql.types as T

spark = SparkSession.builder.getOrCreate()
```

TIP This chapter will not be a deep dive in SQL rather than a comparison with PySpark and a way to combine both. I am leaving certain concepts out on purpose, but it doesn't mean you can't use them if you've comfortable with SQL.

SIDEBAR **ANSI SQL vs. HiveQL**

Spark supports both ANSI SQL and the vast majority of HiveQLfootnote::[You can see the functionality supported (and unsupported on the Spark website: https://docs.databricks.com/spark/latest/spark-sql/compatibility/hive.html] as a SQL dialect. Spark SQL also has some Spark specific functions baked in to ensure a common set of functionality across languages.

In a nutshell, Hive is a SQL-like interface that can be used over a variety of data storage. It became very popular because it allowed to query files in HDFS (Hadoop Distributed File System) like if they were a table. Spark can integrate with Hive when your environment has it installed. Spark SQL also provides additional syntax to work with larger datasets, which I will cover as we need it.

Because of the amount of material and its longevity, and also because they are very similar in syntax for basic and intermediate queries, I usually recommend learning ANSI SQL at first and then complete with HiveQL as you go along. This way, your knowledge will transfer to other SQL-based products.

Understanding Hive in depth is something I won't cover in this book as it is not a component of PySpark. For those who are curious, *Practical Hive* (Apress, 2016) seems like a good reference.

7.1 Banking on what we know: `pyspark.sql` vs plain SQL

PySpark's data frame manipulation hints at its SQL heritage: the name of the module—`pyspark.sql`— is a dead giveaway. PySpark developers recognized the heritage of the SQL programming language for data manipulation and used the same vocabulary to name their method. Let's look at a quick example in both SQL and plain PySpark and look at similarities between the keywords used. In , I load a CSV containing information about the periodic table of elements and I query the data set to find the number of entries with a `liquid` state per period. The code is presented both in PySpark and SQL form, and without much context, we can see similarities between the two.

In , I put the code side by side and illustrate where the similarities start and end. Even though it's a small example, it shows the main differences between PySpark and SQL.

1. PySpark will always start with the name of the data frame you are working with. SQL refers to the table (or *target*) using a `from` keyword;
2. PySpark chains the transformations and actions as methods on the data frame, where SQL splits them into two groups: the *operation* group and the *condition* group. The first one is before the `from` clause and operates on columns. The second is after the from clause and group, filters, and orders the structure of the result table.

TIP SQL is not case sensitive, so you can either use lower-case or upper-case. I usually prototype in lower-case then convert in upper-case when I am confident about my query, to differentiate the two visually.

```
elements = spark.read.csv(
    "../../data/Ch07/Periodic_Table_Of_Elements.csv",
    header=True,
    inferSchema=True,
)

elements.where(F.col("phase") == "liq").groupby("period").count().show()

// In SQL: We assume that the data is in a table called `elements`

SELECT
  period,
  count(*)
FROM elements
WHERE phase = "liq"
GROUP BY period;
```

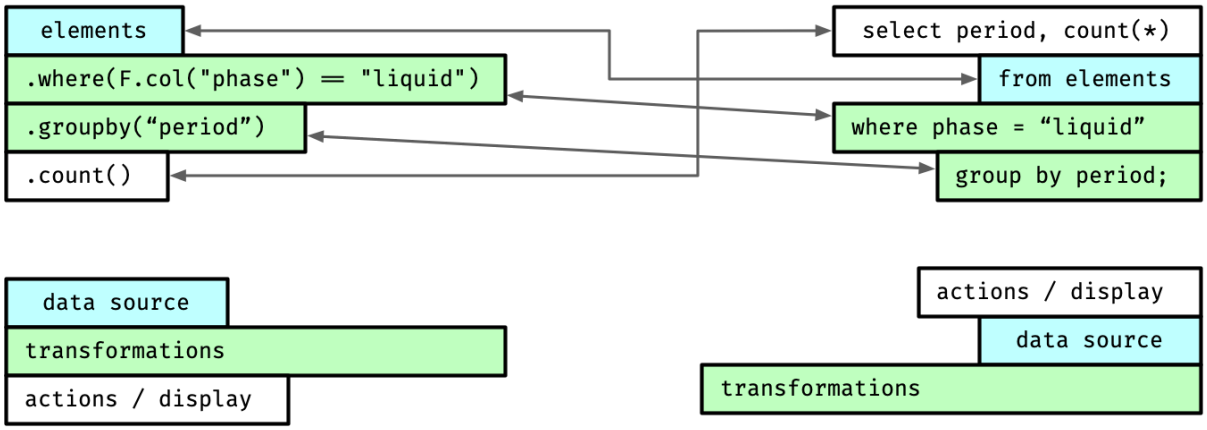


Figure 7.1 The same simple statement, represented in PySpark and SQL

Both would return the same results: one element in period four (Bromine) and one in period six (Mercury).

Whether you prefer the order of operations from PySpark and SQL will depend on how you build query mentally and how familiar you are with the respective language. Fortunately, PySpark makes it easy to move from on to the other, and even work with both all at once.

7.2 Using SQL queries on a data frame

Since we can think of PySpark data frames like tables on steroids, it's not farfetched to think about querying them using a language designed to query tables. Spark provides a full SQL API which is documented in the same fashion as the PySpark one (https://spark.apache.org/docs/latest/api/sql/index.html). We also see that the functions defined in `pyspark.sql.functions` are defined.

> **NOTE** Spark's SQL API only covers the data manipulation subset of Spark. You won't be able to do machine learning using SQL, for instance.

7.2.1 Promoting a data frame to a Spark table

PySpark maintains boundaries between its own namespacing and Spark SQL's namespacing. This means that a data set loaded in PySpark won't be automatically available to querying using SQL. The code in shows an example of this.

Listing 7.1 Trying (and failing) at querying a data frame SQL-style

```
try:
    spark.sql(
        "select period, count(*) from elements where phase='liq' group by period"
    ).show(5)
except AnalysisException as e:
    print(e)

# 'Table or view not found: elements; line 1 pos 29'
```

Here, PySpark doesn't make the link between the python variable `elements`, which points to the data frame, and a potential table `elements` that can be queried by Spark SQL. In order to allow a data frame to be queried via SQL, we need to *register* them as tables. I illustrated the process in . When we assign a data frame to a variable, Python points to the data frame. Spark SQL does not have visibility over the variables Python assigns.

When you want to create a table to query with Spark SQL, you can use the `createOrReplaceTempView()` method. This method takes a single string parameter which is the name of the table you want to use. This transformation will look at the data frame referenced by the Python variable on which the method was applied and will create a Spark SQL reference to the same data frame. We see an example of this in the bottom halr.

> **NOTE** Although you can name the table the same name as the variable you are using, you are not forced to do so. I could have named my table `starneifwpy` if I wanted to.

Once we have registered our `elements` table that point to the same data frame that our Python

variable of the same name, we can query our table without any problem. Let's re-run the same code block as and see that it succeds.

> **NOTE** In this chapter, I use the term "table" and "view" pretty loosely. In SQL, they are distinct concepts: the table is materialized in memory and the view is computed on the fly. Spark's temp views are conceptually closer to a view than a table. Spark SQL also has tables but we will not be using them, preferring reading and materializing our data into a data frame.

Listing 7.2 Trying (and succeding) at querying a data frame SQL-style

```
elements.createOrReplaceTempView("elements")    ❶

spark.sql(
    "select period, count(*) from elements where phase='liq' group by period"
).show(5)

# +------+--------+
# |period|count(1)|
# +------+--------+
# |     6|       1|
# |     4|       1|
# +------+--------+   ❷
```

❶ We register our table using the `createOrReplaceTempView()` method on the `element` data frame.

❷ The same query works once Spark is able to dereference the SQL view name.

So now we have a view registered. In the case of a low number of views to manage, it's pretty easy to keep their name in memory. What about if you have dozens of them or if you need to delete some? Enter the catalog, Spark's way of managing its SQL namespace.

SIDEBAR **Advanced-ish topic: Spark SQL views and persistence**

TL;DR: Use `createOrRepleaceTempView()` as it does what you expect.

PySpark has four methods to create temporary views and they look quite similar at first glance.

- `createGlobalTempView`
- `createOrReplaceGlobalTempView`
- `createOrReplaceTempView`
- `createTempView`

We can see that there is two by two matrix of possibilities:

1. Do I want to replace an existing view (`OrReplace`)?
2. Do I want to create a Global view (`Global`)?

The first one is relatively easy to answer: if you use `createTempView` with a name already being used for another table, the method will fail, where it'll replace the reference if you use `createOrReplaceTempView()`. In SQL, it is equivalent to use `CREATE VIEW` vs. `CREATE OR REPLACE VIEW`. I personally always use the latter as it mimics Python's way of doing things: when re-assigning a variable, you comply.

What about `Global`? The difference between a local view and a global view has to do with how long it will last in memory. A local table is tied to your `SparkSession` where a global table is tied to the Spark application. The differences at this time are not significant at all as we are not using multiple `SparkSession` that need to share data. Because of this, I usually don't use the `Global` methods.

7.2.2 Using the Spark catalog

The Spark catalog is an object that allows working with Spark SQL tables and views. A lot of its methods has to do with managing the metadata of those tables, such as their name and the level of caching (which I'll cover in details in Chapter 9). We will look at the most basic set of functionality here, leaving the more advanced parts, such as caching and user-defined functions, to their respective chapters.

We can use the catalog to list the tables/views we have registered and drop them if we are done. The code in provides the simple methods to do those tasks. Since they are mostly mimicing PySpark's data frame functionality, I think that an example shows it best.

Listing 7.3 Using the catalog to display our registered view and then drop it

```
spark.catalog    ❶

#  <pyspark.sql.catalog.Catalog at 0x117ef0c18>

spark.catalog.listTables()    ❷

#  [Table(name='elements', database=None, description=None,
#         tableType='TEMPORARY', isTemporary=True)]

spark.catalog.dropTempView("elements")    ❸

spark.catalog.listTables()    ❹

# []
```

❶ The catalog is reached through the `catalog` property of our `SparkSession`

❷ The `listTables` method gives us a list of `Table` objects containing the information we want

❸ To delete a view, we use the method `dropTempView()`, passing the name of the view as a parameter

❹ Our catalog now has no table for us to query

How that we understand how we can manage a Spark SQL view within PySpark, we can start looking at manipulating data using both languages.

7.3 SQL and PySpark

The integration between Python (through PySpark) and SQL is very well thought out and can improve the speed at which we can write code. This section will cover the fundamental case, which is using a single language. I will review the most common operations from a pure-SQL and a purist PySpark way, to illustrate how basic manipulations are written.

For the remainder of the chapter, we will be using a public data set provided by BackBlaze which provided hard drive data and statistics. Their data is in the gigabytes range, which although not "big" yet is certainly Spark worthy as it'll be more than the memory available on your home computer. For those willing to scale their program beyond a single computer, I recommend peeking at Appendix B to provision a cloud cluster and use the whole data provided on the website. A convenience shell script is also provided for downloading everything in one fell swoop. For those working locally and afraid of blowing your memory, you can use only Q3 2019. The syntax will differ marginally between both workflows. A 16GB laptop should be able to use all files.

Backblaze provides documentation mostly in the form of SQL statements, which is perfect for what we're learning.

To get the files, you can either download them from the website (https://www.backblaze.com/b2/hard-drive-test-data.html) or use the `backblaze_download_data.py` available in the code repository, which need the `wget` package to be installed. The data needs to be in the `./data/Ch07` directory.

Listing 7.4 Downloading the data from backblaze

```
$ pip install wget

$ python backblaze_download_data.py full

# [some data download progress bars]

$ ls  ../../data/Ch07     ❶

# Periodic_Table_Of_Elements.csv data_Q2_2019.zip              data_Q4_2019.zip
# data_Q1_2019.zip               data_Q3_2019.zip

$ unzip '../../data/Ch07/*.zip'
```

❶ Windows users, use `dir ..\..\data\Ch07`

Make sure you unzip the files into the directory before trying to read them. Unlike many other codecs (Gzip, Bzip2, Snappy and LZO, for instance), PySpark will not decompress zip files automatically when reading them, so we need to do it ahead of time. The `unzip` command can be used if you are using the command line (you might need to install the tool on Linux). On Windows, I usually use the Windows Explorer and unzip by hand.

The code to ingest and prep the data is pretty straightforward. We read each data source separately, and then we make sure that each data frame has the same columns as its peers. In our case, the data for the fourth quarter has two more columns than the others, so we add the missing columns. When unioning the four data frames together, we use a select method on the data frames so their column order is all the same. We continue by casting all the columns containing a SMART measurement as a Long, since they are documented as integral values. Finally, we register our data frame as a view so we can use SQL statements on it.

Listing 7.5 Reading the backblaze data into a data frame and registering a view

```
DATA_DIRECTORY = "../../data/Ch07/"

q1 = spark.read.csv(
    DATA_DIRECTORY + "drive_stats_2019_Q1", header=True, inferSchema=True
)
q2 = spark.read.csv(
    DATA_DIRECTORY + "data_Q2_2019", header=True, inferSchema=True
)
q3 = spark.read.csv(
    DATA_DIRECTORY + "data_Q3_2019", header=True, inferSchema=True
)
q4 = spark.read.csv(
    DATA_DIRECTORY + "data_Q4_2019", header=True, inferSchema=True
)

# Q4 has two more fields than the rest

q4_fields_extra = set(q4.columns) - set(q1.columns)

for i in q4_fields_extra:
    q1 = q1.withColumn(i, F.lit(None).cast(T.StringType()))
    q2 = q2.withColumn(i, F.lit(None).cast(T.StringType()))
    q3 = q3.withColumn(i, F.lit(None).cast(T.StringType()))


# if you are only using the minimal set of data, use this version
backblaze_2019 = q3

# if you are using the full set of data, use this version
backblaze_2019 = (
    q1.select(q4.columns)
    .union(q2.select(q4.columns))
    .union(q3.select(q4.columns))
    .union(q4)
)

# Setting the layout for each column according to the schema

q = backblaze_2019.select(
    [
        F.col(x).cast(T.LongType()) if x.startswith("smart") else F.col(x)
        for x in backblaze_2019.columns
    ]
)

backblaze_2019.createOrReplaceTempView("backblaze_stats_2019")
```

7.4 Using SQL-like syntax within data frame methods

Our goal in this section is to perform a quick exploratory data analysis on a subset of the columns presented. We will reproduce the failure rates that Backblaze computes themselves and identify the models with the most and least amount of failures in 2019.

7.4.1 Select and where

Those operations are like the bread and butter of data manipulation, yet we already see a difference in how SQL orders the operations differently than PySpark. The code in explores a few models that have failed. This is useful as a quick-and-dirty way to see if the data looks consistent and see the serial numbers nomenclature.

Listing 7.6 Comparing `select` and `where` in PySpark and SQL

```
spark.sql(
    "select serial_number from backblaze_stats_2019 where failure = 1"
).show(
    5
)    ❶

backblaze_2019.where("failure = 1").select(F.col("serial_number")).show(5)

# +-------------+
# |serial_number|
# +-------------+
# |    57GGPD9NT|
# |     ZJV02GJM|
# |     ZJV03Y00|
# |     ZDEB33GK|
# |     Z302T6CW|
# +-------------+
# only showing top 5 rows
```

❶ Since a SQL statement returns a data frame, we still have to `show()` it to see the results.

PySpark makes you think about how you want to chain the operations. In our case, we start by filtering the data frame and then we select the column of interest. SQL presents an alternative construction:

1. You put the columns you want to select at the beginning of your statement. This is called the *SQL operation*
2. You them add one or more tables to query, called the *target*
3. After, you add the *conditions*, such as filtering.

Every operation we will look in this chapter will be classified as an operation, a target, or a condition, so you can know where it fits in the statement.

> **TIP** If you have a table you want to extract as a data frame, you can just assign the result of a `SELECT` statement to variable.

7.4.2 Group and order by

Here, we are looking at the capacity in gigabytes of the hard drives included in the data, by model. For this, we use a little bit of arithmetic and the `pow()` function, that elevates its first argument to the power of the second. We can see similarities about the SQL and PySpark vocabulary, but once again the order of the transformations is different.

Listing 7.7 Grouping and ordering in PySpark and SQL

```
spark.sql(
    """SELECT
           model,
           min(capacity_bytes / pow(1024, 3)) min_GB,
           max(capacity_bytes/ pow(1024, 3)) max_GB
        FROM backblaze_stats_2019
        GROUP BY 1
        ORDER BY 3 DESC"""
).show(5)

backblaze_2019.groupby(F.col("model")).agg(
    F.min(F.col("capacity_bytes") / F.pow(F.lit(1024), 3)).alias("min_GB"),
    F.max(F.col("capacity_bytes") / F.pow(F.lit(1024), 3)).alias("max_GB"),
).orderBy(F.col("max_GB"), ascending=False).show(5)

# +--------------------+--------------------+-------+
# |               model|              min_GB| max_GB|
# +--------------------+--------------------+-------+
# |       ST16000NM001G|             14902.0|14902.0|
# | TOSHIBA MG07ACA14TA|-9.31322574615478...|13039.0|
# |HGST HUH721212ALE600|             11176.0|11176.0|
# |       ST12000NM0007|-9.31322574615478...|11176.0|
# |       ST12000NM0008|             11176.0|11176.0|
# +--------------------+--------------------+-------+
# only showing top 5 rows
```

In PySpark, once again, we look at the logical order of operations. We `groupby` the `capacity_GB` columns. which is a computed column. Just like in PySpark, arithmetic operations can be performed using the usual syntax in SQL. Furthermore, the `pow()` function— available in `pyspark.sql.functions`— is also implemented in Spark SQL. If you need to see which functions can be used out of the box, the Spark SQL API doc contains the necessary information (https://spark.apache.org/docs/latest/api/sql/index.html).

Grouping and ordering are conditions in SQL, so they are at the end of the statement. One thing worth noting is that we group by `1` and order by `3 DESC`. This is a short-hand way of referring to the columns in the SQL operation by position rather than name. In this case, it saves us from writing `group by capacity_bytes / pow(1024, 3)` or `order by max(capacity_bytes / pow(1024,3)) DESC` in the conditions block.

Looking at the results from our query, there are some drives that report more than one capacity. Furthermore, we have some drives reporting negative capacity, which is really odd. Let's focus on seeing how prevalent this is.

7.4.3 Having

Let's assume that, for each model, the maximum reported capacity is the correct one. Because of how SQL is evaluated, we can't refer to an aliased field in our `WHERE` clause. Because of this, we have to rely to another keyword in order to compare our min and max reported capacity. The code in shows how we can accomplish this in both languages.

Listing 7.8 Using `having` in SQL, and relying on `where` in PySpark

```
spark.sql(
    """SELECT
           model,
           min(capacity_bytes / pow(1024, 3)) min_GB,
           max(capacity_bytes/ pow(1024, 3)) max_GB
        FROM backblaze_stats_2019
        GROUP BY 1
        HAVING min_GB != max_GB
        ORDER BY 3 DESC"""
).show(5)

backblaze_2019.groupby(F.col("model")).agg(
    F.min(F.col("capacity_bytes") / F.pow(F.lit(1024), 3)).alias("min_GB"),
    F.max(F.col("capacity_bytes") / F.pow(F.lit(1024), 3)).alias("max_GB"),
).where(F.col("min_GB") != F.col("max_GB")).orderBy(
    F.col("max_GB"), ascending=False
).show(
    5
)

# +--------------------+--------------------+-------+
# |               model|              min_GB| max_GB|
# +--------------------+--------------------+-------+
# | TOSHIBA MG07ACA14TA|-9.31322574615478...|13039.0|
# |       ST12000NM0007|-9.31322574615478...|11176.0|
# |HGST HUH721212ALN604|-9.31322574615478...|11176.0|
# |       ST10000NM0086|-9.31322574615478...| 9314.0|
# |HGST HUH721010ALE600|-9.31322574615478...| 9314.0|
# +--------------------+--------------------+-------+
# only showing top 5 rows
```

Having is a syntax unique to SQL: it can be thought of a `where` clause that can only be applied to aggregate fields, such as `count(*)` or `min(date)`. Since it's equivalent in functionality to where, `having` is in the condition block after the `group by` clause. In PySpark, we do not have `having` as a method. Since each method returns a new data frame, we do not have to have a different keyword, and can just use `where` with the column we created instead.

We will ignore (for now) those capacity reporting inconsistencies. They'll come back as exercises.

7.4.4 Create tables/views

Now that we have queried the data and are getting the hang of it in SQL, we might want to checkpoint our work and save some data so we do not have to process everything from scratch the next time. For this, we can either create a table of a view, which we'll then be able to query directly.

Creating a table or a view is very easy in SQL: we just have to prefix our query by `CREATE TABLE/VIEW`. Here, creating a table or a view will have a different impact. If you have a Hive metastore connected, creating a table will materialize the data (for more information, see

appendix B) where a view will only keep the query. To take a baking analogy, `CREATE TABLE` will store a cake, where `CREATE VIEW` will refer to the ingredients (the original data) and the recipe (the query).

To demonstrate this, I will reproduce the `drive_days` and `failures` that compute the number of days of operation a model has and the number of drive failures it has had, respectively. The code in shows how it is done: you prefix your select query with a `CREATE [TABLE/VIEW]`.

In PySpark, we do not have to rely on extra syntax. A newly created data frame just has to be assigned to a variable and we are good to go.

Listing 7.9 Creating a view in Spark SQL and in PySpark

```
spark.sql(
    """
    CREATE VIEW drive_days AS
        SELECT model, count(*) AS drive_days
        FROM drive_stats
        GROUP BY model"""
)

spark.sql(
    """CREATE VIEW failures AS
           SELECT model, count(*) AS failures
           FROM drive_stats
           WHERE failure = 1
           GROUP BY model"""
)

drive_days = backblaze_2019.groupby(F.col("model")).agg(
    F.count(F.col("*")).alias("drive_days")
)

failures = (
    backblaze_2019.where(F.col("failure") == 1)
    .groupby(F.col("model"))
    .agg(F.count(F.col("*")).alias("failures"))
)
```

SIDEBAR **Creating tables from data in SQL**

You can also create a table from data on a hard drive or HDFS. For this, you can use a modified SQL query like so. Since we are reading a CSV file, we prefix our path by `csv.`.

```
spark.sql("create table q1 as select *
    from csv.`../../Data/Ch07/drive_stats_2019_Q1`")
```

I much prefer relying on PySpark syntax for reading and setting the schema from my data source and then using SQL, but the option is there for the taking.

7.4.5 Union and join

So far, we've seen how to query a single table at a time. In practice, you'll often get multiple tables related to one another. We already witnessed this problem by having one historical table per quarter, which needed to be stacked (or unioned) together, and with our `drive_days` and `failures` tables, which each paint a single dimension of the story until they are merged (or joined) together.

Joins and unions are the only clauses we'll see that are modifying the target piece in our SQL statement. In SQL, a query is operating on a single target at a time. We already saw at the beginning of the chapter how to use PySpark to union tables together. In SQL, we follow the same blueprint: `SELECT columns FROM table1 UNION ALL SELECT columns FROM table2`

WARNING **PySpark's `union()` SQL `UNION`**

In SQL, `UNION` removes the duplicate records. PySpark's `union()` doesn't, which is why it's equivalent to a SQL `UNION ALL`. If you want to drop the duplicates, which is an expensive operation when working in a distributed context, use the `distinct()` function after your `union()`. This is one of the rare case where PySpark's vocabulary doesn't follow SQL's, but it's for a good reason. Most of the time, you'll want the `UNION ALL` behaviour.

It is always a good idea to make sure that your data frames have the same columns, with the same types, in the same order, before attempting a union. In the PySpark solution, we used the fact that we can extract the columns in a list to `select` the data frames in the same fashion. Spark SQL does not have a simple way to do the same, so one would have to type all the columns. It's alright when you just have a few, but we're talking hundred here.

One easy way to circumvent this is to use the fact that a Spark SQL statement is a string. We can take our list of columns, transform it into a SQL-esque string and be done with it. This is exactly what I did in . It's not a pure Spark SQL solution, but it's much friendlier than making you type all the columns one by one.

WARNING Do not allow for plain string insertion if you are processing user input! This is the best way to have a SQL injection, where a user can craft a string that will wreck havok on your data. For more information about SQL injections and why, have a look at https://owasp.org/www-community/attacks/SQL_Injection.

Listing 7.10 Unioning tables together in Spark SQL and in PySpark

```
columns_backblaze = ", ".join(q4.columns)    ❶

q1.createOrReplaceTempView("Q1")    ❷
q1.createOrReplaceTempView("Q2")
q1.createOrReplaceTempView("Q3")
q1.createOrReplaceTempView("Q4")

spark.sql(
    """
    CREATE VIEW backblaze_2019 AS
    SELECT {col} FROM Q1 UNION ALL
    SELECT {col} FROM Q2 UNION ALL
    SELECT {col} FROM Q3 UNION ALL
    SELECT {col} FROM Q4
""".format(
        col=columns_backblaze
    )
)

backblaze_2019 = (    ❸
    q1.select(q4.columns)
    .union(q2.select(q4.columns))
    .union(q3.select(q4.columns))
    .union(q4)
)
```

❶ We use the `join()` method on a separator string to create a string containing all the elements in the list, separated by `, `

❷ We promote our quarterly data frames to Spark SQL views so we can use them in our query

❸ This is taken from

Join are equally as simple in SQL. We add a `[DIRECTION] JOIN table [ON] [LEFT COLUMN] [OP] [RIGHT COLUMN]` in the target portion of our statement. The direction is the same parameter of our `how` in PySpark. The `on` clause is a series of comparison between columns. In the example, we are joining the records where the value in the `model` column is equal (`=`) on both `drive_days` and `failures` tables.

Listing 7.11 Joining the `drive_days` and `failures` tables together, in Spark SQL and in PySpark

```
spark.sql(
    """select
           drive_days.model,
           drive_days,
           failures
    from drive_days
    left join failures
    on
        drive_days.model = failures.model"""
).show(5)

drive_days.join(failures, on="model", how="left").show(5)
```

7.4.6 Subqueries and common table expressions

The last piece of SQL syntax we will look on its own is the subquery and the common table expression. A lot of SQL references do not talk about them until very late which is a shame because they are a) easy to understand and b) are very helpful in keeping your code clean. In a nutshell, they allow to create tables that are local to your query. In Python, this would be similar to using the `with` statement or using a function block to limit the scope of a query. I will show the function approach as it is much more commonfootnote::[The `with` statement is usually used with resources that need to be cleaned up at the end. It doesn't really apply here, but I felt like the comparison was worth mentionning.].

For our example, we will take our `drive_days` and `failures` table definitions and bundle them into a single query that will measure the models with the highest rate of failure in 2019. The code in shows how we can do this using a subquery. A subquery simply replace a table name with a stand-alone SQL query. In the example, we can see that the name of the table has been replaced by the `SELECT` query that formed the table. We can alias the "table" referred in the subquery by adding the name at the end of the statement, after the closing parenthesis.

Listing 7.12 Finding the drive models with the highest rate of failure using subqueries

```
spark.sql(
    """
    SELECT
        model,
        failures / drive_days failure_rate
    FROM (
        SELECT
            model,
            count(*) AS drive_days
        FROM drive_stats
        GROUP BY model) drive_days
    INNER JOIN (
        SELECT
            model,
            count(*) AS failures
        FROM drive_stats
        WHERE failure = 1
        GROUP BY model) failures
    ON
        drive_days.model = failures.model
    ORDER BY 2 desc
    """
).show(5)
```

Subqueries are cool but can be hard to read and debug, since you are adding some complexity into the main query. This is where common table expressions, or CTE, are especially useful. A CTE is a table definition, just like in the subquery case. The difference here is that you put them at the top of your main statement (before your main `SELECT`) and prefix with the word `WITH`. In ,

I am taking the same statement as with the subquery case but using two CTE instead. They can also be considered makeshift `CREATE` statements that get dropped at the end of the query, just like the `with` keyword in Python.

Listing 7.13 Finding the drive models with the highest rate of failure using common table expressions

```
spark.sql(
    """
    WITH drive_days as (
        SELECT
            model,
            count(*) AS drive_days
        FROM drive_stats
        GROUP BY model),
    failures as (
        SELECT
            model,
            count(*) AS failures
        FROM drive_stats
        WHERE failure = 1
        GROUP BY model)
    SELECT
        model,
        failures / drive_days failure_rate
    FROM drive_days
    INNER JOIN failures
    ON
        drive_days.model = failures.model
    ORDER BY 2 desc
    """
).show(5)
```

In Python, the best comparison I've found was to wrap our statements in a function. Any intermediate variable created in the scope of the function would not be kept once the function returns. My version of the query using PySpark is in .

Listing 7.14 Finding the drive models with the highest rate of failure using Python scope rules to approximate CTE.

```
def failure_rate(drive_stats):
    drive_days = drive_stats.groupby(F.col("model")).agg(    ❶
        F.count(F.col("*")).alias("drive_days")
    )

    failures = (
        drive_stats.where(F.col("failure") == 1)
        .groupby(F.col("model"))
        .agg(F.count(F.col("*")).alias("failures"))
    )
    answer = (    ❷
        drive_days.join(failures, on="model", how="inner")
        .withColumn("failure_rate", F.col("failures") / F.col("drive_days"))
        .orderBy(F.col("failure_rate").desc())
    )
    return answer


failure_rate(drive_stats).show(5)

print("drive_days" in dir())    ❸
```

❶ We are creating intermediate data frames within the body of the function to avoid having a monster query.

❷ Our `answer` data frame uses both intermediate data frames

❸ Testing if we have a variable `drive_days` in scope once the function returned confirms that our intermediates frames are neatly confined inside the function scope.

This section took a (very) simple and high level EDA and demonstrated how we can do it using PySpark or Spark SQL. PySpark gives the floor to SQL without too much ceremony. This can be very convenient if you happen to hang out with DBAs and SQL developpers, as you can collaborate using their preferred language, while knowing that Python is right around the corner. Everybody wins!

7.4.7 A quick summary of PySpark vs. SQL syntax

PySpark borrowed a lot of vocabulary from the SQL world. I think this was a very smart idea: there are generations of programmers that know SQL, and adopting the same keywords Python-style makes it easy to communicate. Where we see a lot of difference is in the order of the operations: PySpark will naturally encourage you to think about the order of which the operations should be performed. SQL follows a more rigit framework where you need to remember if your operation belong in the operation, the target or the condition clause.

I find PySpark's way of treating data manipulation more intuitive, but will rely on my years of SQL experience as a data analyst when convenient. When writing SQL, I usually write my query out order, starting with the target and building as I go. Not everything needs to be top to bottom!

So far, I've made a lot of effort to keep both languages in a vacuum. We'll break the barrier now and unleash the power of Python+SQL. This will simplify how we write certain transformations, make our code easier to write and a less busy.

7.5 Simplifying our code: blending SQL and Python together

PySpark is rather accommodating when taking method and function parameters: you can pass a column name instead of a column object (`F.col()`) when using `groupby()`, for instance. On top of this, there are a few methods we can use to cram a little SQL syntax into our PySpark code. You'll see that there isn't many methods where you can use this, but it's so useful and well done that you'll end up using it all the time.

This section will build on the code we've written so far. We're going to write a function that, for a given capacity, will return the top 3 most reliable drives according to our failure rate. We'll leverage the code written so far and simplify it.

7.5.1 Reading our data

We start by simplifying the code to read the data. The data ingestion part of the program is displayed in . Compared to our original data ingestion, I've made a few changes.

First, I've put all the directories in a list so I could read them using a list comprehension. This removes some repetitive code and will also work easily if I remove or add some files (if you were only using Q3 2019, you can remove the other entries in the list).

Second, since we do not need the SMART measurements, I've taken the intersection of the columns instead of trying to fill the missing columns with `null` values. In order to create a common intersection that would apply to any number of data sources, I've used `reduce` which applies the anonymous function on all the column sets, resulting in the common set of columns between all the data frames. For those unfamiliar with `reduce` and other higher-order functions, Appendix D contains a deeper explanation of how they works. I have also added an assertion on the common set of columns, as I want to make sure it contains the columns I need for the analysis. Assertions are a good way to short circuit an analysis if certains conditions are not met. In this case, if I am missing one of the colums, I'd rather have my program fail early with an `AssertionError` rather than have a huge stack trace later. Assertions are covered in detail in Chapter 13.

Finally, I have used a second `reduce` for unioning all the distinct data frames in a cohesive one. The same principle is used as when I was creating the common set of variables. This makes the code a lot cleaner and it will work without any modifications should I want to add more sources or remove some.

Listing 7.15 The data ingestion part of our program

```
from functools import reduce

import pyspark.sql.functions as F
from pyspark.sql import SparkSession

spark = SparkSession.builder.getOrCreate()

DATA_DIRECTORY = "../../data/Ch07/"

DATA_FILES = [
    "drive_stats_2019_Q1",
    "data_Q2_2019",
    "data_Q3_2019",
    "data_Q4_2019",
]

data = [
    spark.read.csv(DATA_DIRECTORY + file, header=True, inferSchema=True)
    for file in DATA_FILES
]

common_columns = list(
    reduce(lambda x, y: x.intersection(y), [set(df.columns) for df in data])
)

assert set(["model", "capacity_bytes", "date", "failure"]).issubset(
    set(common_columns)
)

full_data = reduce(
    lambda x, y: x.select(common_columns).union(y.select(common_columns)), data
)
```

7.5.2 Using SQL-style expressions in PySpark

Now that our data has been read and is in a steady state, we can process it so we have an easy time answering our question. I will use SQL-style expressions when appropriate to showcase when it makes sense to fuse both languages. At the end of this section, we'll have code that will:

1. Select only the useful columns for our query
2. Get our drive capacity in gigabytes
3. Compute the `drive_days` and `failures` data frames
4. Join the two data frames into a summarized one and compute the failure rate

The code is available in .

Listing 7.16 Processing our data so it's ready for the query function

```
full_data = full_data.selectExpr(
    "model", "capacity_bytes / pow(1024, 3) capacity_GB", "date", "failure"
)

drive_days = full_data.groupby("model", "capacity_GB").agg(
    F.count("*").alias("drive_days")
)

failures = (
    full_data.where("failure = 1")
    .groupby("model", "capacity_GB")
    .agg(F.count("*").alias("failures"))
)

summarized_data = (
    drive_days.join(failures, on=["model", "capacity_GB"], how="left")
    .fillna(0.0, ["failures"])
    .selectExpr("model", "capacity_GB", "failures / drive_days failure_rate")
    .cache()
)
```

SQL-style expression, one that we already have been using. One of the new ones is `selectExpr()` which is just like the `select()` method with the exception that it will process SQL-style operations. I am quite a fan of this since it removes a bit of syntax when manipulating columns with functions and arithmetic. In our case, the PySpark alternative (displayed in) is a little more verbose and cumbersome to write and read, especially since we have to create a literal `1024` column to apply the `pow()` function.

Listing 7.17 Replacing `selectExpr()` by a regular `select()` in our final program

```
full_data = full_data.selectExpr(
    F.col("model"),
    (F.col("capacity_bytes") / F.pow(F.lit(1024), 3)).alias("capacity_GB"),
    F.col("date"),
    F.col("failure")
)
```

The second one is simply called `expr()`. It wraps a SQL-style expression into a column. This is kind of a generalized `selectExpr()`, where you can use it in lieu of of `F.col()` (or the column name) when you want to modify a column. If we take our `failures` table from , we can use an `expr` (or *expression*) as the `agg()` argument. This alternative syntax is shown in . I like doing it in `agg()` parameters, because it saves a lot of `alias()`.

Listing 7.18 Using a SQL `expr`ession in our `failures` data frame code.

```
failures = (
    full_data.where("failure = 1")
    .groupby("model", "capacity_GB")
    .agg(F.expr("count(*) failures"))
)
```

The third one, and my favourite, is the `where()/filter()` method. Filtering conditions in SQL

is something I am very familiar with and being able to use them directly in PySpark with no ceremony is a godsend. In our final program, I was able to use `full_data.where("failure = 1")` instead of having to wrap the column name in `F.col()` like we've been doing since the beginning of this book.

I re-use this convenience in the query function, which is displayed in . This time, I've used string interpolation in conjunction with between. This doesn't save many keystokes, but it's very easy to understand and you don't get as much line noise as when using the `data.capacity_GB.between(capacity_min, capacity_max)` or (if you prefer using the column function) `F.col("capacity_GB").between(capacity_min, capacity_max)`. At this point, it's very much a question of personal style and how familiar you are with each approach. I recommend you to try the SQL one if you don't have a favourite yet.

Listing 7.19 The `most_reliable_drive_for_capacity()` function, that computes the the top N drives for a given capacity

```
def most_reliable_drive_for_capacity(data, capacity_GB=2048, precision=0.25, top_n=3):
    """Returns the top 3 drives for a given approximate capacity.

    Given a capacity in GB and a precision as a decimal number, we keep the N
    drives where:

    - the capacity is between (capacity * 1/(1+precision)), capacity * (1+precision)
    - the failure rate is the lowest

    """
    capacity_min = capacity_GB / (1 + precision)
    capacity_max = capacity_GB * (1 + precision)

    answer = (
        data.where(f"capacity_GB between {capacity_min} and {capacity_max}")    ❶
        .orderBy("failure_rate", "capacity_GB", ascending=[True, False])
        .limit(top_n)    ❷
    )

    return answer
```

❶ I used an SQL-style expression in my `where()` method, without having to use any other special syntax or method

❷ Since we want to return the top N results, not just show them, I use `limit()` instead of `show()`.

7.6 Conclusion

You do not need to learn or use SQL to effectively work with PySpark. That being said, since the data manipulation API shares so much vocabulary and functionality with SQL makes it a plus to at least have a basic understanding of the syntax and query structure.

My family speaks both English and French, and sometimes you don't always know where one language starts and one ends. I tend to think in both languages, and sometimes blends them in a single sentence. Likewise, I find that some problems are easier to solve with Python and some

are more in SQL's territory. You will find your own balance as well, which is why it's nice to have the option. Just like spoken languages, the goal is to express your thoughts and intentions as clearly as possible, keeping your audience in mind.

7.7 Summary

- Spark provides an SQL API for data manipulation. This API supports ANSI SQL.
- Spark (and PySpark's by extension) borrows a lot of vocabulary and expected functionality from the way SQL manipulates tables. This is especially evident since the data manipulation module is called `pyspark.sql`.
- PySpark's data frames need to be registered as views or tables before they can be queried with Spark SQL. You can give them a different name than the data frame you're registering.
- PySpark's own data frame manipulation methods and functions borrow SQL functionality for the most part. Some exceptions, such as `union()`, are present and documented in the API.
- Spark SQL queries can be inserted in a PySpark program through the `spark.sql` function, where `spark` is the running `SparkSession`.
- Spark SQL tables references are kept in a `Catalog` which contains the metadata for all tables accessible to Spark SQL.
- PySpark will accept SQL-style clauses in `where()`, `expr()` and `selectExpr()`, which can simplify the syntax for complex filtering and selection.
- When using Spark SQL queries with user-provided input, be careful about sanitizing the inputs to avoid potential SQL injection attacks.

7.8 Exercises

7.8.1 Exercise 7.1

If we look at the code in , we can simplify it even further, avoiding the creation of the two tables outright. Can you write a summarized_data without having to use another table than `full_data` and no join? (Bonus: try using pure PySpark, then pure Spark SQL, then a combo of both.)

7.8.2 Exercise 7.2

Our analysis is a flawed in that the age of a drive is not taken into consideration. Instead of ordering the drives by failure rate, order them by average age at failure (assume that every drive fails on 2020-01-01 if they are still alive at the end of the year).

7.8.3 Exercise 7.3

What is the total capacity (in TB) that BackBlaze records at the end of each month?

7.8.4 Exercise 7.4

Note: We will revisit this exercise in Chapter 10 when we look at window functions. In the meantime, this can be solved with a judicious usage of group bys and joins.

If you look at the data, you'll see that some drive models can report an erroneous capacity. In the data preparation stage, use the most common capacity for each drive so we do not have more than one entry in our function.

8 *Extending PySpark with user-defined-functions*

This chapter covers:

- How to use the RDD as a low level, flexible data container.
- How to promote regular Python functions to UDF to run in a distributed fashion.
- How to use scalar UDF as an alternative to Python UDF, using pandas' API.
- How to use grouped map and grouped aggregate UDF on `GroupedData` object to split data frame computation on manageable chunks.
- How to apply UDF on local data to ease debugging.

Our journey with PySpark so far has proven that it is a powerful and versatile data processing tool. So far, we've explored many out-of-the-box functions and methods to manipulate data in a data frame. PySpark's data frame manipulation functionality takes our Python code and applies an optimized query plan, introduced in Chapter 1. This makes our data jobs efficient, consistent, and predictable, just like coloring within the lines. What if we need to go off-script and manipulate our data according to our own rules?

In this chapter, I cover how we can build Python functions and scale them in PySpark. I start by introducing the resilient distributed dataset (or RDD), a more primitive and lower-level structure compared to the data frame. I explain how you manipulate data in an RDD and how its element (or row) major nature complements the data frame column-major approach.

I build on the knowledge of the RDD and explain how we can transfer this to the data frame through user-defined functions (or UDF). Following this, I move to pandas UDF, a speedy and powerful way to distribute Python functions on data frames. Finally, I close the loop by discussing how to use Scala modules in PySpark, so your programs can build on the shoulders of others. With those additional tools at your disposal, no data manipulation task will leave you stumped!

This chapter uses pandas from 8.3 onwards, for pandas UDF. Extensive pandas knowledge is a nice-to-have but is in no way expected. This chapter will cover the necessary pandas skills to use in within a basic pandas UDF. If your wish to level up you pandas skills to become a pandas UDF ninja, I warmly recommend the *Pandas in Action* book, by Boris Parkhaver (Manning, 2021).

For the examples in the chapter, we will need three previously unused libraries: pandas, scikit-learn, and PyArrow. If you have installed Anaconda (see appendix B), you can use `conda` to install the libraries; otherwise, you can use `pip` [7].

```
# Conda installation
conda install pandas sklearn pyarrow

# Pip installation
pip install pandas sklearn pyarrow
```

8.1 PySpark, freestyle: the resilient distributed dataset

This section covers the resilient distributed dataset (RDD) and how you use it to manipulate data. My goal is to provide a good overview of what an RDD is and how you manipulate data using Python functions. Besides being useful in itself, this section's material is a perfect introduction to user-defined functions, the data frame's answer to the RDD operating model. I cover user-defined functions in the following sections.

The RDD can be thought of as the ultimate flexible data container structure. You can cram pretty much anything that can be pickled (python's way of serializing objects) into it. Where the data frame has good structure documentation though `Column` objects, types, and schemas, the RDD does not force you into a specific layout. Each element is independent of the other. In listing 8.1, I create a list containing multiple objects of different types, then promote it to an RDD via the `parallelize` method. The resulting RDD is depicted in figure 8.1.

Listing 8.1 Promoting a Python list to a resilient distributed dataset

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.getOrCreate()

collection = [1, "two", 3.0, ("four", 4), {"five": 5}]  ❶

sc = spark.sparkContext  ❷

collection_rdd = sc.parallelize(collection)  ❸

print(collection_rdd)
# ParallelCollectionRDD[0] at parallelize at PythonRDD.scala:195  ❹
```

❶ My `collection` is a list of an integer, a string, a float, a tuple, and a dict.

❷ The RDD functions and methods are under the `SparkContext` object, accessible as an attribute of our `SparkSession`. I alias it to `sc` for convenience.

❸ The list gets promoted to an RDD using the `parallelize` method of the `SparkContext`

❹ Our `collection_rdd` object is effectively an RDD. PySpark returns the type of the collection when we print the object.

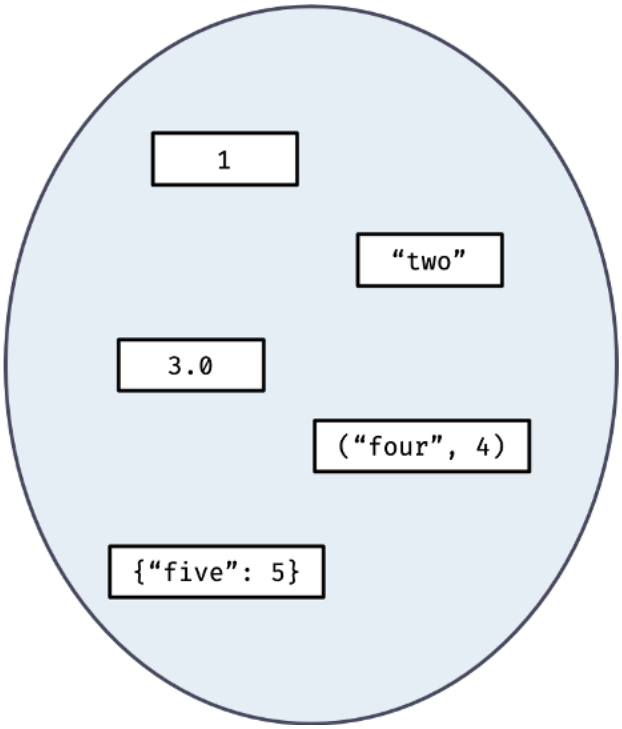


Figure 8.1 The colletion_rdd RDD. Each object is independent from each other in the container. No column, no structure, no schema.

Compared to a data frame, the RDD is much more *free style* (pardon the 90's reference) in terms of what it accepts. Manipulating data follows the same pattern: instead of using transformations on column objects, we go straight to the data, element by element. The next section introduces the RDD data manipulation API.

8.1.1 Manipulating data the RDD way: map, filter and reduce

This section explains the building blocks of data manipulation using an RDD. I discuss the concept of *higher-order function* and we use them to transform data. I finish with a quick overview of MapReduce, a fundamental concept in large-scale data processing, and place in the context of Spark and the RDD.

Manipulating data with an RDD feels to me like giving orders to an army as a general: you have full obedience from your privates/divisions into the field/RDD, but if you give an incomplete or wrong order, you'll cause havoc within your troop. Furthermore, each division has its own,

specific type of order they can perform, and you don't have a reminder of what's what (unlike with a data frame schema). Sounds like a fun job…

An RDD provides many methods (which you can find in the API documentation for the `pyspark.RDD` object), but we put our focus on three specific methods: `map()`, `filter()` and `reduce()`. Together, they capture the ethos of data manipulation with an RDD; knowing how those three work will give you the necessary foundation to understand the others.

`map()`, `filter()` and `reduce()` all take a function (that we will call `f`) as their only parameter. We call functions that take other functions as parameters *higher-order functions*. They can be a little difficult to understand if it's the first time you are encountering them; fear not, after seeing them in action, you'll be very comfortable using them in PySpark (and in Python, if you have a look at appendix D).

MAP

`map()` is the easiest one to figure out: it applies the function taken as a parameter to every element of the RDD. I try, and fail, to map a simple function that adds 1 to a value in listing 8.2. What's happening?

Listing 8.2 Mapping a simple function `add_one()` to each element to an RDD.

```
def add_one(value):
    return value + 1  ❶


collection_rdd = collection_rdd.map(add_one)  ❷

print(collection_rdd.collect())  ❸
# Stack trace galore! The important bit, you'll get one of the following:
# TypeError: can only concatenate str (not "int") to str
# TypeError: unsupported operand type(s) for +: 'dict' and 'int'
# TypeError: can only concatenate tuple (not "int") to tuple
```

❶ A seeminly inoffensive function; `add_one()` adds one to the value passed as an argument.

❷ I apply my function to every element in the RDD, via the `map()` method.

❸ `collect()` materializes an RDD into a Python list on the master node.

To understand why our code is failing, we'll break down the mapping process, illustrated in figure 8.2. I apply the `add_one()` function to each element in the RDD by passing it as an argument to the `map()` method. `add_one()` is a regular Python function, applied to regular Python objects. Since we have incompatible types (taking an example, `"two" + 1` is not a legal operation in Python), three of our elements are `TypeError`. When I `collect()` the RDD to peek at the values, it explodes into a stack trace right in my REPL.

NOTE The RDD is a lazy collection. If you have an error in your function application, it will not be visible until you perform an action (such as `collect()`), just like with the data frame.

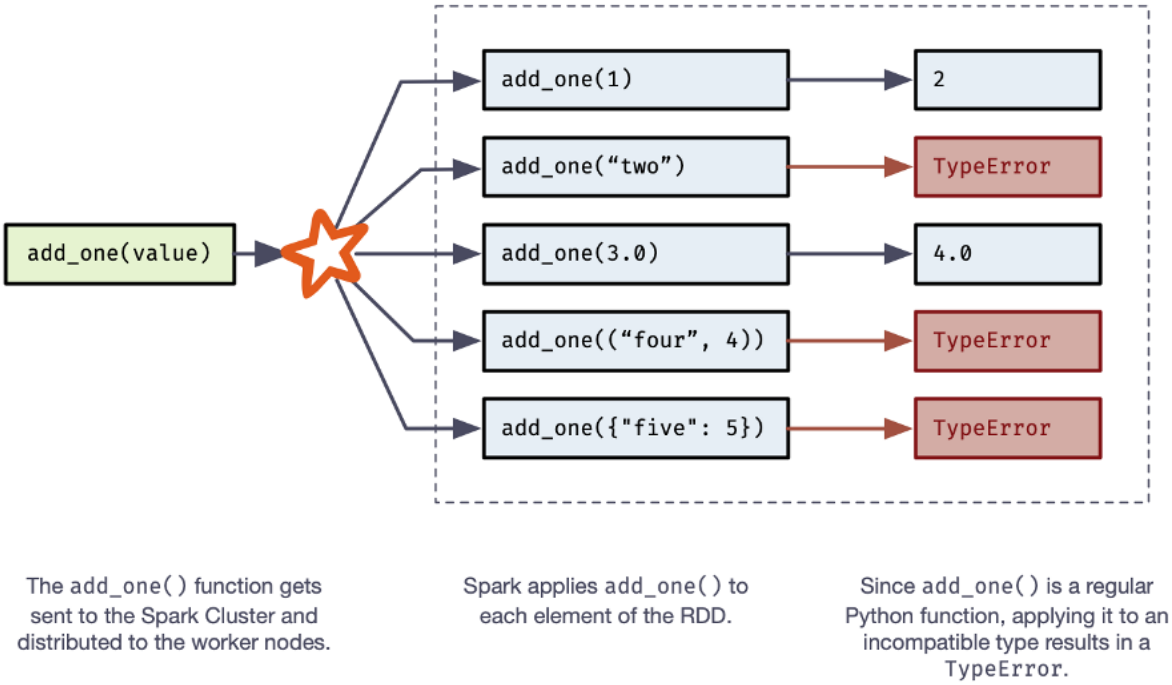


Figure 8.2 Applying the add_one() function to each element of the RDD via map. If the function cannot be applied, an error will be raised during action time.

Fortunately, since we are working with Python, we can use a `try/except` block to prevent errors. I provide an improved `safer_add_one()` function in listing 8.3, which returns the original element if the function runs into a type error.

Listing 8.3 Mapping the function `safer_add_one()` to each element to an RDD. Each element that can't be incremented will remain the same in the resulting RDD.

```
collection_rdd = sc.parallelize(collection)  ❶


def safer_add_one(value):
    try:
        return value + 1
    except TypeError:
        return value  ❷


collection_rdd = collection_rdd.map(safer_add_one)

print(collection_rdd.collect())
# [2, 'two', 4.0, ('four', 4), {'five': 5}] ❸
```

❶ I recreate my RDD from scratch to remove the erreneous operation in the thunk (see chapter 1 for a description of a computation thunk).

❷ Our function returns the original value untouched if it encounters a `TypeError`

❸ The relevant elements of the RDD have been incremented by one.

In summary, you use `map()` to apply a function to every element of the RDD. Because of the flexibility of the RDD, PySpark does not give you any safeguards about the content of the RDD. You are responsible, as the developer, to make your function robust regardless of the input.

FILTER

`filter()` is used to keep only the element that satisfies a predicate. In the data frame world, we encountered the `where()/filter()` methods that do just that. In the RDD world, `filter()` is much more flexible: it takes a function `f` and keeps only the elements that return a truthful value. In listing 8.4, I filter my RDD to keep only the integer and float elements, using a lambda function. The `isinstance()` function returns `True` if the first argument's type is present in the second argument; in our case, it'll test if each element is either a `float` or an `int`.

Listing 8.4 Filtering our RDD with a lambda function. Our resulting RDD only has `int` and `float` values.

```
collection_rdd = collection_rdd.filter(lambda elem: isinstance(elem, (float, int)))

print(collection_rdd.collect())
# [2, 4.0]
```

Just like `map()`, the function passed as a parameter to `filter()` is applied to every element in the RDD. This time, though, instead of returning the result in a new data frame, we keep the original value if the result of the function is truthy. If the result is falsey, we drop the element. I show the breakdown of the `filter()` operation in figure 8.3.

SIDEBAR **Truthy/Falsey in Python**

Python has its own rules for boolean testing: because of this, I avoid using absolute True/False when talking about filtering in Python and PySpark. As a rule of thumb, `False`, `0` (the number zero in any Python numerical type), and empty sequences and collections (list, tuple, dict, set, range) are falsey. For more precisions on how Python imputes boolean values on non-booleans types, refer straight to the Python documentation: docs.python.org/3/library/stdtypes.html#truth-value-testing.

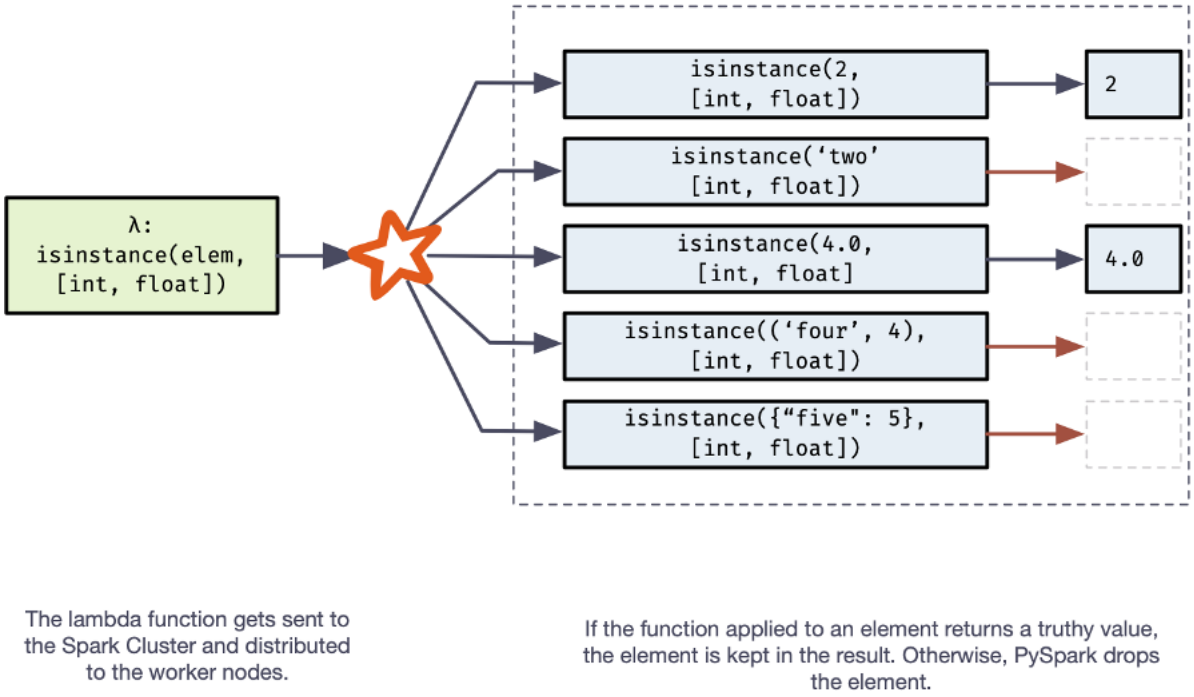


Figure 8.3 Filtering our RDD to keep only int and float. Our predicate function is applied element-wise, and only the values leading to truthy predicates are kept.

REDUCE

The last operation is `reduce()` and as its name implies, it is being used to reduce elements in an RDD. By reducing, I mean taking 2 elements and applying a function that will return only one element. PySpark will apply the function to the first two elements, then apply it again to the result and the third element, and so on until there are no elements left. I find the concept easier when explained visually, so figure 8.4 shows the process of summing the value of the elements in the RDD using reduce. In code form, listing 8.5 presents how to use the `reduce()` method on a data frame.

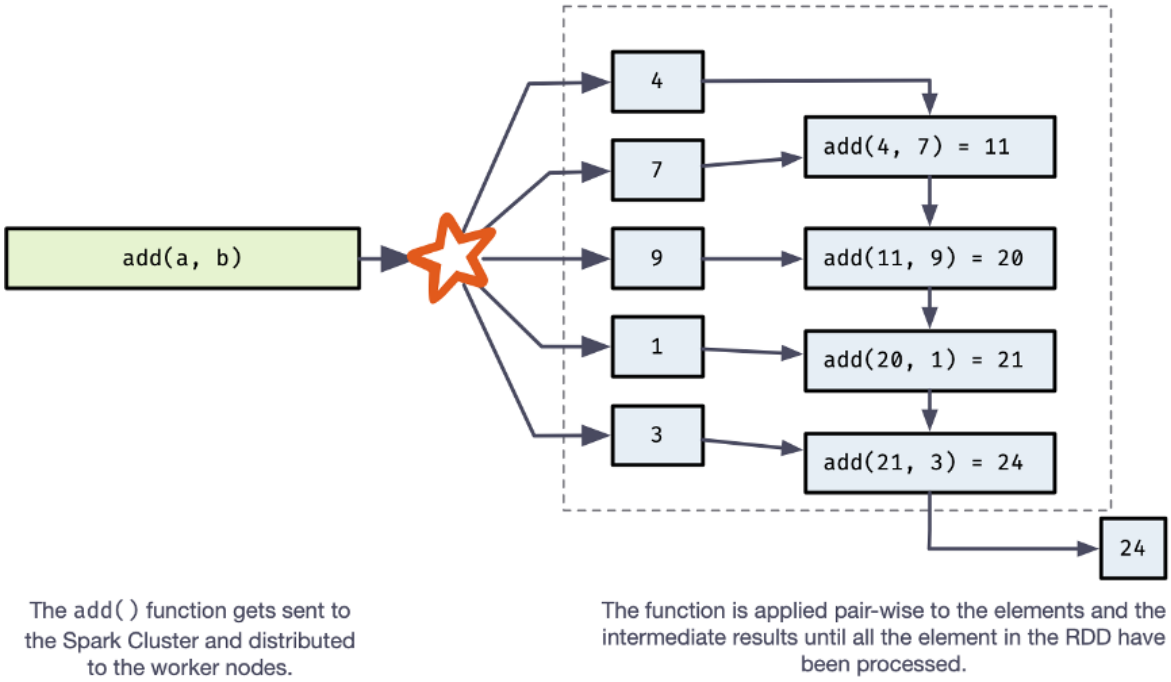


Figure 8.4 Reducing our RDD by summing the values of the elements.

Listing 8.5 Applying the `add()` function via `reduce()` to get the sum of the values of the elements of our RDD.

```
from operator import add  ❶

collection_rdd = sc.parallelize([4, 7, 9, 1, 3])

print(collection_rdd.reduce(add))  # 24
```

❶ The `operator` module contains the function version or common operators such as `+` (`add()`), so we do not have to pass a `lambda a, b: a + b`.

`map()`, `filter()`, and `reduce()` appear at a first glance like simple concept: they take a function and apply it to all the elements inside the collection. The result is treated differently depending on the method chosen, and `reduce()` requires a function of two arguments returning a single value. Well, in 2004, Google used this humble concept and caused a revolution in the large-scale data processing world by publishing its MapReduce framework (research.google/pubs/pub62/). You can't argue on inspiration: the name is a combination of `map()` and `reduce()`! This framework was a direct inspiration to big data frameworks such as Hadoop and Spark. Although modern abstractions such as the data frame aren't as close to the original MapReduce, the ideas remain, and by understanding at a high level the building blocks, it'll be easier to understand some higher-level design choices.

SIDEBAR **`reduce()` in a distributed world**

Because of PySpark's distributed nature, the data of an RDD can be distributed across multiple partitions. The `reduce()` function will be applied independently on each partition, and then each intermediate value will be sent to the master node for the final reduction. Because of this, you need to provide a *commutative* and *associative* function to `reduce()`.

A *commutative function* is a function in which you do not care about the order in which the arguments are applied. For example, `add()` is commutative, since `a + b = b + a`. Oh the flip side, `subtract()` is not: `a - b != b - a`.

An *associative* function is a function in which you do not care about how the values are grouped. `add()` is associative, since `(a + b) + c = a + (b + c)`. `subtract()` is not: `(a - b) - c != a - (b = c)`.

`add()`, `multiply()`, `min()` and `max()` are both associative and commutative.

This concludes our whirlwind tour of the PySpark RDD API. We covered how the RDD applies transformations to its elements through higher-order functions such as `map()`, `filter()`, and `reduce()`. Those higher-order functions apply the functions passed as parameters to each element, making the RDD "element-major" (or "row-major"). If you are curious about the other applications of the RDD, I recommend looking at Appendix F as a companion to the PySpark online API documentation. Most of the methods on the RDD have a direct equivalent to the data frame or grow directly from the usage of `map()`, `filter()`, and `reduce()`. We revisit the RDD briefly in chapter 9 when talking about performance. The next sections build on the concept of applying Python functions directly, but this time on a data frame. The fun only begins!

EXERCISE **8.1**

The PySpark RDD API provides a `count()` method that returns the number of elements in the RDD as an integer. Reproduce the behavior of this method using `map()`, `filter()` and/or `reduce()`.

EXERCISE **8.2**

What is the return value of the following code block?

```
a_rdd = sc.parallelize([0, 1, None, [], 0.0])

a_rdd.filter(lambda x: x).collect()
```

1. `[1]`
2. `[0, 1]`
3. `[0, 1, 0.0]`
4. `[]`
5. `[1, []]`

SIDEBAR **Optional topic: going full circle, a data frame is an RDD!**

To show the ultimate flexibility of the RDD, have a look at this: you can access an implicit RDD within a data frame via the `rdd` attribute of a data frame.

Listing 8.6 Uncovering the RDD from within a data frame using the `rdd` attribute

```
df = spark.createDataFrame([[1], [2], [3]], schema=["column"])

print(df.rdd)
# MapPartitionsRDD[22] at javaToPython at NativeMethodAccessorImpl.java:0

print(df.rdd.collect())
# [Row(column=1), Row(column=2), Row(column=3)]
```

From a PySpark perspective, a data frame is also an `RDD[Row]` (from `pyspark.sql.Row`), where each row can be thought of a dictionary: the key is the column name and the value is the value contained in the record. To do the opposite trip, you can pass the RDD to `spark.createDataFrame` with an optional schema. Remember that, when moving from a data frame to an RDD, you give up the schema safety of the data frame!

It can be tempting to move back and forth between a data frame and an RDD depending on the operation you wish to perform. Bear in mind that this will come at a performance cost (which we'll explore further in chapter 9), but also makes your code harder to follow. You will also have to make sure all your `Row` follow the same schema before promoting your RDD into a data frame. The next sections cover how you can harness most of the power of the RDD without leaving the comfort of the data frame.

8.2 Using Python to extend PySpark via user-defined functions

In the previous section, we got a taste of flexibility with the RDD approach to data manipulation. This section takes the same question — how can we run Python code on our data? — and applies it to the data frame. More specifically, we focus on the `map()` transformation: for each record that comes in, one record comes out. Map-type transformations are by far the most frequent and the easiest to implement.

Unlike the RDD, the data frame has structure enforced by columns. To address this constraint, PySpark provides the possibility to create *user-defined functions* via the `pyspark.sql.functions.udf()` function. What comes in is a regular Python function, and out is a function promoted to work on PySpark columns.

To illustrate this, we will mock up a data type not present in PySpark: the `Fraction`. Fractions are made of a numerator and a denominator. In PySpark, we'll represent this as an array of two integers. In listing 8.7, I create a data frame containing two columns, standing for the numerator and the denominator. I fuse the two columns in an array column via the `array()` function.

Listing 8.7 Creating a data frame containing a single array column, where the first element is the numerator and the second the denominator.

```
import pyspark.sql.functions as F
import pyspark.sql.types as T

fractions = [[x, y] for x in range(100) for y in range(1, 100)]  ❶

frac_df = spark.createDataFrame(fractions, ["numerator", "denominator"])

frac_df = frac_df.select(
    F.array(F.col("numerator"), F.col("denominator")).alias("fraction"),  ❷
)

frac_df.show(5, False)
# +--------+
# |fraction|
# +--------+
# |[0, 1]  |
# |[0, 2]  |
# |[0, 3]  |
# |[0, 4]  |
# |[0, 5]  |
# +--------+
# only showing top 5 rows
```

❶ I start the range for the denominator at 1, since a fraction with 0 for the denominator is undefined

❷ The `array()` function takes two or more columns of the same type and creates a single column containing an array of the columns passed as parameter.

To support our new makeshift fraction type, we create a few functions that provide basic functionality. This is a perfect job for Python UDF, and I take the opportunity to introduce the

two ways PySpark enables its creation.

8.2.1 It all starts with plain Python: using typed Python functions

This section covers creating a Python function that will work seamlessly with a PySpark data frame. While Python and Spark are like PB & J, creating and using UDF requires a few precautions. I introduce how you can use Python type hints to make sure your code will work seamlessly with PySpark types. At the end of this section, we will have a function to reduce a fraction and one to transform a fraction into a floating-point number.

My blueprint when creating a function destined to become a Python UDF is as follow:

1. Create and document the function
2. Make sure the input and output types are compatible
3. Test the function

Testing PySpark code (including UDF) is covered in chapter 14. For this section, I provide a couple of assertions to make sure the function is behaving like expected.

Behind every UDF is a Python function, so our two functions are in listing 8.8. I introduce Python type annotations in this code block: the rest of the section covers how they are used in this context and why they are a powerful tool when combined with Python UDF.

Listing 8.8 Creating our three python functions, complete with type annotation and assertions

```
from fractions import Fraction  ❶
from typing import Tuple, Optional  ❷

Frac = Tuple[int, int]  ❸


def py_reduce_fraction(frac: Frac) -> Optional[Frac]:  ❹
    """Reduce a fraction represented as a 2-tuple of integers."""
    num, denom = frac
    if denom:
        answer = Fraction(num, denom)
        return answer.numerator, answer.denominator
    return None


assert py_reduce_fraction((3, 6)) == (1, 2)  ❺
assert py_reduce_fraction((1, 0)) is None


def py_fraction_to_float(frac: Frac) -> Optional[float]:
    """Transforms a fraction represented as a 2-tuple of integers into a float."""
    num, denom = frac
    if denom:
        return num / denom
    return None


assert py_fraction_to_float((2, 8)) == 0.25
assert py_fraction_to_float((10, 0)) is None
```

❶ We rely on the `Fraction` data type from the `fractions` module to avoid reinventing the wheel

❷ Some specific types need to be imported to be used: the standard library contains the types for scalar values, but containers like `Option` and `Tuple` need to be explicitly imported.

❸ We create a type synonym `Frac`. This is equivalent to telling Python/mypy "*When you see* `Frac` *, assume it's a* `Tuple[int, int]` " (a tuple containing two integers). This makes the type annotations easier to read.

❹ Our function takes a `Frac` as argument and returns a `Optional[Frac]`, which translates to "*either a* `Frac` *or* `None` ".

❺ I create a few assertions to sanity check my code and make sure I get the expected behavior.

Both functions are very similar, so I'll take `py_reduce_fraction` and go through it line by line.

My function definition has a few new elements. The `frac` parameter has a `: Frac` and we have a `Optional[Frac]` before the colon. Those additions are *type annotations* and are an amazing tool in making sure the function accepts and returns what we expect. Python is a dynamic language: this means that the type of an object is known at runtime. When working with PySpark's data frame, where each column has one and only one type, we need to make sure that our UDF will return consistent types. We can use type hints to ensure this.

Python's type checking is enabled by using a library called mypy. You install it via `pip install mypy`. Once installed, you can run mypy on your file with `mypy MY_FILE.py`. Appendix D contains a deeper introduction to the `typing` module and mypy and how it applies (and why it should apply) beyond UDF. I'll add type annotation when relevant, as they can be useful documentation besides making our code more robust. (*What does my function expect? What does it return?*)

In my function definition, I announce that the `frac` function parameter is of type `Frac`, which is equivalent to a `Tuple[int, int]`, or a 2-element tuple containing two integers. If I get to share my code with others, this *type annotation* sends a signal about the input type of my function. Furthermore, mypy will complain if I try to pass an incompatible argument to my function: if I try to do `py_reduce_fraction("one half")`, mypy will tell me the following.

```
error: Argument 1 to "py_reduce_fraction" has incompatible type "str"; expected "Tuple[int, int]"
```

I can already see the type errors vanishing…

The second type annotation, located after the function arguments and prefixed with an arrow, is the type annotation for the return type of the function. We recognize the `Frac`, but this time, I

wrapped it into an `Optional` type.

In 8.1, when creating functions to be distributed over the RDD, I needed to make sure that they would not trigger an error, returning `None` instead. I apply the same concept here. I test for `denom` being a truthy value: if it is equal to 0, I return `None`. This is such a frequent use-case that Python provides the `Optional[…]` type, which means "either the type between the brackets or `None` ". PySpark will accept `None` values as `null` (see appendix D for the complete list of Python vs. Spark types).

SIDEBAR **Type annotations: stop cluttering my code!**

Type annotations are incredibly useful out of the box, but they are especially nifty when used with Python UDF. Since PySpark's execution model is lazy, you'll often get your error stack trace at action time. UDF stack traces are not any harder to read than any other stack trace in PySpark — which is not saying much — but a vast majority of the bugs are because of a bad input or return value. Now, type annotations are not a silver bullet, but they are a great tool to avoid and diagnose type errors.

With all this said, Python's type annotations are not available everywhere. When working with pandas, numpy and PySpark's data structures, you might encounter "*no stub file for [module]*". Python's typing story is still early, and you might run into some rough edges. Fortunately, mypy tries to minimize the annoyance by only checking the annotated functions, and you can search for "data science types" on PyPI for an interim solution.

The rest of the function is relatively straightforward: I ingest the numerator and denominator in a `Fraction` object, which reduces the fraction. I then extract the numerator and denominator from the `Fraction` and return them as a tuple of 2 integers, as I promised in my return type annotation.

We have our two functions, with well-defined input and output types. In the next section, I show how you promote regular Python functions to UDF and apply them to your data frame.

8.2.2 From Python function to UDF: two approaches

PySpark provides two equivalent ways to promote a function to a UDF. In this section, I explain how to use both approaches. I also discuss how to provide return Python type hints when creating a UDF.

PySpark provides a `udf()` function in the `pyspark.sql.functions` module to promote Python function to their UDF equivalent. The function takes two parameters.

1. The function you want to promote.

2. Optionally, the return type of the generated UDF. In table 8.1, I summarize the type equivalences between Python and PySpark. If you provide a return type, it *must* be compatible with the return value of your UDF.

Table 8.1 A summary of the types in PySpark. A star next to the "Python equivalent" column means the Python type is more precise or can contain larger values, so you need to be careful with the values you return.

Type Constructor	String representation	Python equivalent
`NullType()`	`null`	`None`
`StringType()`	`string`	Python's regular strings
`BinaryType()`	N/A	bytearray
`BooleanType()`	`boolean`	bool
`DateType()`	`date`	`datetime.date` (from the `datetime` library)
`TimestampType()`	`timestamp`	`datetime.datetime` (from the `datetime` library)
`DecimalType(p,s)`	`decimal`	`decimal.Decimal` (from the `decimal` library)*
`DoubleType()`	`double`	float
`FloatType()`	`float`	float*
`ByteType()`	`byte` or `tinyint`	int*
`IntegerType()`	`int`	int*
`LongType()`	`long` or `bigint`	int*
`ShortType()`	`short` or `smallint`	int*
`ArrayType(T)`	N/A	list, tuple or Numpy array (from the `numpy` library)
`MapType(K, V)`	N/A	dict
`StructType([…])`	N/A	list or tuple

In listing 8.9, I promote the `py_reduce_fraction()` function to a UDF via the `udf()` function. Just like I did with the Python equivalent, I provide a return type to the UDF (this time, an `Array` of `Long`, since `Array` is the companion type of the tuple and `Long` the one for Python integers). Once the UDF is created, we can apply it like any other PySpark function on columns. I chose to create a new column to showcase the before and after: in the sample shown, the fraction appears properly reduced.

Listing 8.9 Creating a UDF explicitly with the `udf()` function, and applying it to our data frame.

```
SparkFrac = T.ArrayType(T.LongType())  ❶

reduce_fraction = F.udf(py_reduce_fraction, SparkFrac)  ❷

frac_df = frac_df.withColumn(
    "reduced_fraction", reduce_fraction(F.col("fraction"))  ❸
)

frac_df.show(5, False)
# +--------+----------------+
# |fraction|reduced_fraction|
# +--------+----------------+
# |[0, 1]  |[0, 1]          |
# |[0, 2]  |[0, 1]          |
# |[0, 3]  |[0, 1]          |
# |[0, 4]  |[0, 1]          |
# |[0, 5]  |[0, 1]          |
# +--------+----------------+
# only showing top 5 rows
```

❶ I alias the "array of long" PySpark type to the `SparkFrac` variable.

❷ I promote my Python function using the `udf()` function, passing my `SparkFrac` type alias as the return type.

❸ A UDF can be used like any other PySpark column function

You also have the option to create your Python function and promote it as a UDF using the `udf` function as a decorator. In listing 8.10, I define by `py_fraction_to_float()` (now called simply `fraction_to_float()`) directly as a UDF by preceding my function definition by `@F.udf([return_type])`. In both cases, you can access the underlying function from the UDF by calling the attribute `frac`.

Listing 8.10 Creating a UDF directly using the `udf()` decorator.

```
@F.udf(T.DoubleType())  ❶
def fraction_to_float(frac: Frac) -> Optional[float]:
    """Transforms a fraction represented as a 2-tuple of integers into a float."""
    num, denom = frac
    if denom:
        return num / denom
    return None


frac_df = frac_df.withColumn(
    "fraction_float", fraction_to_float(F.col("reduced_fraction"))
)

frac_df.select("reduced_fraction", "fraction_float").distinct().show(5, False)
# +----------------+-------------------+
# |reduced_fraction|fraction_float     |
# +----------------+-------------------+
# |[3, 50]         |0.06               |
# |[3, 67]         |0.04477611940298507|
# |[7, 76]         |0.09210526315789473|
# |[9, 23]         |0.391304347826087  |
# |[9, 25]         |0.36               |
# +----------------+-------------------+
# only showing top 5 rows
assert fraction_to_float.func((1, 2)) == 0.5  ❷
```

❶ The decorator performs the same function as the `udf()` function, but return a UDF bearing the name of the function defined under.

❷ In order to perform my assertion, I use the `func` attribute of the UDF, which returns the function ready to be called.

EXERCISE 8.3

Create a UDF that adds two fractions together, and test it by adding the `reduced_fraction` to itself in the `frac_df` data frame.

EXERCISE 8.4

Our `py_reduce_fraction` will not work if the numerator or denominator exceeds `pow(2, 63)-1` or is lower than `-pow(2, 63)`. Modify the `py_reduce_fraction` to return `None` if this is the case.

Bonus: Does this change the type annotation provided? Why?

8.3 Big data is just a lot of small data: using pandas UDF

> **NOTE** Spark 3.0 brings a ton of improvements and new functionality to pandas UDF, including scalar iterator, map iterator, and cogrouped map pandas UDF. The chapter is written with Spark 2.4.5, the latest available version, in mind, but I plan on including a new section containing the new Spark 3.0 UDF (and refresh the current material if necessary) once I get them under the microscope.

Python UDF, while very flexible, only operates on a single record at a time, just like the `map()` method of the RDD. This section introduces a fresh way to approach UDF: pandas (or *vectorized*) UDF. Just like their name indicates, they rely on pandas, a very popular data manipulation library in Python.

> **WARNING** Vectorized UDF were introduced in Spark 2.3 (scalar, grouped map) and improved upon in Spark 2.4 (grouped aggregate). I recommend using the most up-to-date stable version everywhere in the book, but this section requires it.

At the core, pandas UDF can be seen as nothing more than distributing pandas data manipulation code within a data frame. In chapter 1, I explained that Spark distributes a large amount of data in multiple partitions and orchestrates transformations and actions through a master-workers split. In our model, a pandas UDF is just like treating each chunk of data like an independent pandas data frame.

8.3.1 Setting our environment: connectors and libraries

For this section, I use the National Oceanic and Atmospheric Administration (NOAA) Global Surface Summary of the Day (GSOD) dataset. This data is available from multiple sources, but one of the easiest to access is Google pubic data set repository, made available through BigQuery. I use the BigQuery connector to Spark to ingest the data (github.com/GoogleCloudDataproc/spark-bigquery-connector). The instructions on their Github might change over time, so refer to their README as necessary. For Spark 2.4.5, you will need to download the `spark_bigquery_latest.jar`. If you are using Spark 3.0, you will need the jar for your Scala version (2.11 or 2.12, depending on your Spark installation).

To access the data, you also need a Google Cloud Platform (GCP) account. Once your account is created, you need to create a service account and a service account key to tell BigQuery to give you access to the public data programmatically. To do so, select "Service Account" (under "IAM & Admin") and click "+ Create Service Account". Give a funny name to your service account name. In the service account permissions menu, select "BigQuery → BigQuery admin" and click

"continue". In the last step, click "+ CREATE KEY" and select JSON. Download the key and store it somewhere safe.

> **WARNING** Treat this key like any other password. If a malicious person steals your key, go back to the "Service Accounts" menu and delete this key, recreating a new one.

The last step before analyzing our data using vectorized UDF is to install PyArrow. PyArrow is the python bindings to the Apache Arrow project (arrow.apache.org/), an in-memory data serialization library. It provides a bridge between the PySpark data frame and the pandas data frame. If you are using Spark 2.3 or 2.4, you also need to set a flag in the `conf/spark-env.sh` file of your Spark root directory. In the `conf/` directory, you should find a `spark-env.sh.template` file. Make a copy, name it `spark-env.sh` and add this line in the file.

```
ARROW_PRE_0_15_IPC_FORMAT=1
```

This will tell PyArrow to use a serialization format compatible with Spark 2.X, instead of the newer one only compatible with Spark 3.0. The Spark JIRA ticket contains more information about this (issues.apache.org/jira/browse/SPARK-29367). You can also use PyArrow version 0.14 and avoid the problem altogether.

> **TIP** If you are using PySpark in the cloud, refer to your provider documentation. Each cloud provider has a different way of managing Spark dependencies and libraries. For a quick review of the most popular Spark cloud offerings, see Appendix C.

Finally, we can (re-)start our PySpark shell, with the new library installed. The `pyspark` and `spark-submit` commands take an optional `--jars` parameter that loads external dependencies on your Spark installation.

Listing 8.11 Starting a PySpark shell with the BigQuery connector installed

```
pyspark --jars spark-bigquery-latest.jar
```

Alternatively, if you use PySpark through your Python/IPython shell, you can load the library directly from Maven (Java/Scala's equivalent of PyPI) when creating your `SparkSession`.

Listing 8.12 Initializing PySpark withing your python shell with the BigQuery connector enabled.

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.config(
    "spark.jars.packages",
    "com.google.cloud.spark:spark-bigquery-with-dependencies_2.11:0.15.1-beta",  ❶
).getOrCreate()

# Ivy Default Cache set to: /Users/jonathan_rioux/.ivy2/cache
# The jars for the packages stored in: /Users/jonathan_rioux/.ivy2/jars
# :: loading settings :: url = jar:file:/usr/local/Cellar/apache-spark/2.4.5/libexec/jars/ivy-2.4.0.jar!
# com.google.cloud.spark#spark-bigquery-with-dependencies_2.11 added as a dependency
# :: resolving dependencies :: org.apache.spark#spark-submit-parent-035f1392-cda4-4935-a62b-969bda5449d5
#     confs: [default]
#     found com.google.cloud.spark#spark-bigquery-with-dependencies_2.11;0.15.1-beta in central
# :: resolution report :: resolve 134ms :: artifacts dl 2ms
#     :: modules in use:
#     com.google.cloud.spark#spark-bigquery-with-dependencies_2.11;0.15.1-beta from central in [default]
#     ---------------------------------------------------------------------
#     |                  |            modules            ||   artifacts   |
#     |       conf       | number| search|dwnlded|evicted|| number|dwnlded|
#     ---------------------------------------------------------------------
#     |      default     |   1   |   0   |   0   |   0   ||   1   |   0   |
#     ---------------------------------------------------------------------
# :: retrieving :: org.apache.spark#spark-submit-parent-035f1392-cda4-4935-a62b-969bda5449d5
#     confs: [default]
#     0 artifacts copied, 1 already retrieved (0kB/4ms)
# [...]
```

❶ I took the package version recommended for the most recent Spark/Scala version (2.4.5/2.11)

WARNING If you have a SparkSession already in progress, it is **not enough** to just `spark.stop()` and try to restart. You need to stop the JVM process altogether. Trying to do this without restarting your PySpark/Python REPL is an exercise in frustration, so just kill and start fresh. If you use the method in listing 8.12 and you don't see similar jar verbiage, it will not work.

8.3.2 Preparing our data

Before we can start working on our pandas UDF, we need to extract the data from BigQuery and assemble the multiple tables in a cohesive data frame. Reading data from BigQuery is straightforward. I use the `bigquery` specialized `SparkReader` — provided by the connector library we embedded to our PySpark shell — providing two options:

1. The `table` parameter, pointing to the table we want to ingest. The format is `project.dataset.table`: the `bigquery-public-data` is a project available to all.
2. The `credentialsFile` is the JSON key downloaded in 8.3.1. You need to adjust the path and file name accordingly to the location of the file.

TIP If you are using Google DataProc, you do not have to provide a `credentialsFile` since the permissions will be granted through your GCP account. The documentation for the BigQuery connector will provide the most up-to-date instructions. Appendix C covers Spark in the cloud, including Google DataProc.

The code is available in listing 8.13. For my `gsod` table, I need to union the tables together in a single cohesive data frame. While I can chain multiple `union()` like in chapter 7, I went a more elegant route using the `reduce` operator, this time by applying it to my list comprehension.

Listing 8.13 Reading the `stations` and `gsod` tables for 2010 to 2020.

```
from functools import reduce
from pyspark.sql import DataFrame


def read_df_from_bq(year):  ❶
    return (
        spark.read.format("bigquery")  ❷
        .option("table", f"bigquery-public-data.noaa_gsod.gsod{year}")  ❸
        .option("credentialsFile", "bq-key.json")  ❹
        .load()
    )


gsod = (
    reduce(
        DataFrame.union, [read_df_from_bq(year) for year in range(2010, 2020)]  ❺
    )
    .dropna(subset=["year", "mo", "da", "temp"])
    .where(F.col("temp") != 9999.9)
)
```

❶ Since all the tables are read the same way, I abstract my reading routine in a re-usable function, returning the resulting data frame.

❷ I use the `bigquery` specialized reader via the `format()` method.

❸ The stations table is available in BigQuery under `bigquery-public-data.noaa_gsod.gsodXXXX`, where `XXXX` is the four-digit year.

❹ I pass my JSON service account key to the `credentialsFile` option, to tell Google I am allowed to use the BigQuery service.

❺ `DataFrame.union` can be passed as a parameter to `reduce`, where it'll union all the tables in my list pair-wise into a single table.

It's easier to understand the reduce operation if we break it down into discrete steps.

I start with a **range of years** (in my example 2010 to 2020, including 2010 but excluding 2020).

For this, I use the `range()` function.

I apply my helper function `read_df_from_bq()` to each year via a list comprehension, yielding a list of data frames. I don't have to worry about memory consumption as the list contains only a reference to the data frame (`DataFrame[…]`).

As a reducing function, I use the `DataFrame.union` function. This method, when applied to a data frame (`df.union()`), takes a single parameter, since there is an implicit `self` that maps to the data frame calling the method. If we apply the function *statically*, using it from the generic `DataFrame` object, then it'll take *two data frames as parameter* and union the data frames in a single one.

We could do this iteratively, using a `for` loop. In listing 8.14, I show how to accomplish the same goal without using `reduce()`. Since higher-order functions usually yield cleaner code, I prefer using them to looping constructs where it make sense.

Listing 8.14 Reading the gsod data from 2010 to 2020 using an iterative/looping approach. We need to load a first table to initialize the process.

```
gsod_alt = read_df_from_bq(2010)  ❶
for year in range(2011, 2020):
    gsod_alt = gsod_alt.union(read_df_from_bq(year))
```

❶ When using a looping approach to union tables, you need an explicit starting seed. I use the table from 2010.

The reduce approach only works if all the tables union without any problem; same schema, from column name, order, and types. Google is doing us a solid here by having the data all pre-cleaned for us

> **TIP** If you are using a local Spark, loading 2010-2019 will make the examples in this chapter rather slow. I use 2018 only when working on my local instance so I don't have to wait too long for code execution. Inversely, if you are working with a more powerful setup, you can add years to the range. The `gsod` tables go back to 1929.

8.3.3 Scalar UDF

Scalar UDF are the most common type of pandas UDF. As their name indicates, they work on scalar values: for each record passed in, it needs to return one record. They behave just like regular Python UDF, with one key difference. Python UDF work on one record at a time and you express your logic through regular Python code. Scalar UDF work on one *series* at a time and you express your logic through *pandas* code. The difference is subtle and it's easier to explain visually.

In a Python UDF, when you pass column objects to you UDF, PySpark will unpack each value, perform the computation, and then return the value for each record in a `Column` object. In a Scalar UDF, depicted in figure 8.5, PySpark will serialize (through PyArrow) each partitioned column into a `pandas Series` object (pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.html). You then perform the operations on the Series object directly, returning a Series of the same dimension from your UDF. From an end-user perspective, they are the same functionally In Chapter 9, I discuss the performance implications of Python UDF vs. a scalar UDF.

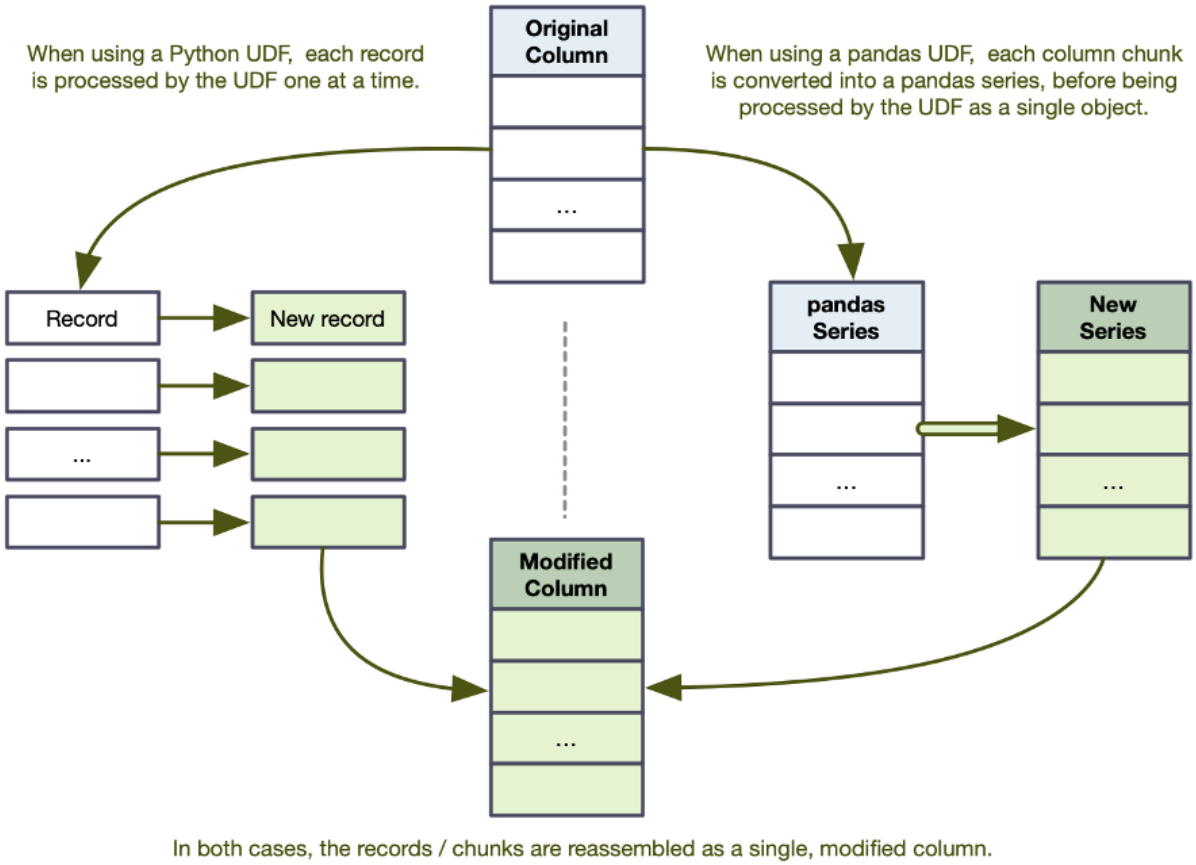


Figure 8.5 Comparing a Python UDF to a pandas scalar UDF. The former splits a column in individual records, where the latter breaks them in Series.

> **NOTE** PySpark makes no guarantees about how the columns you pass to your scalar UDF will be split in Series, so you need to make sure your UDF doesn't depend on a specific breakdown. For more control over the breaks, see listing 8.17.

Now armed with the *how it works?* of scalar UDF, let's create one ourselves. I chose to create a simple function that will transform Fahrenheit degrees to Celcius. In Canada, we use both scales

depending on the usage: F for cooking, C for body or outside temperature. I have no idea of if 95 degrees F is hot or cold, but I know how to dress when it's 10 degrees C, yet my dinner cooks at 350 F.

My function is depicted in listing 8.15. The building blocks are eerily similar; there are two main differences noticeable.

1. Instead of `udf()`, I use `pandas_udf()`, again, from the `pyspark.sql.functions` module. The first parameter is the return type of the function (`DoubleType()`) and the second is an indicator of the type of pandas UDF I am creating, here a `PandasUDFType.SCALAR`.
2. My code itself could be used as-is for a regular python UDF. I am (ab)using the fact that you can do arithmetic operations with pandas Series. You can use any Series method should you need to.

Listing 8.15 Creating a pandas scalar UDF that transforms Fahrenheit into Celcius. I use the `pandas_udf` decorator with a UDF type of `PandasUDFType.SCALAR`.

```
import pandas as pd


@F.pandas_udf(T.DoubleType(), F.PandasUDFType.SCALAR)  ❶
def f_to_c(degrees):
    """Transforms Farhenheit to Celcius."""
    return (degrees - 32) * 5 / 9


f_to_c.func(pd.Series(range(32, 213)))  ❷
# 0        0.000000
# 1        0.555556
# 2        1.111111
# 3        1.666667
# 4        2.222222
#           ...
# 176     97.777778
# 177     98.333333
# 178     98.888889
# 179     99.444444
# 180    100.000000
# Length: 181, dtype: float64

gsod = gsod.withColumn("temp_c", f_to_c(F.col("temp")))
gsod.select("temp", "temp_c").distinct().show(5)

# +-----+------------------+
# | temp|            temp_c|
# +-----+------------------+
# | 37.2| 2.8888888888888906|
# | 85.9| 29.944444444444443|
# | 53.5| 11.944444444444445|
# | 71.6| 21.999999999999996|
# |-27.6|-33.111111111111114|
# +-----+------------------+
# only showing top 5 rows
```

❶ For scalar UDF, the biggest change happens in the decorator used. I could use the `pandas_udf` function directly too.

❷ To test my function, I apply it to every Fahrenheit value from 32 to 212 inclusively (0 to 100 Celcius), using the `func` attribute that returns the local version of the UDF.

Scalar UDF, just like Python regular UDF, are very convenient when the record-wise transformation (or "mapping") you want to apply to your data frame is not available within the stock PySpark functions (`pyspark.sql.functions`). Creating a "Fahrenheit to Celcius" converter as part of core Spark would be a little intense, so using PySpark or (pandas) scalar UDF is a way to extend the core functionality with a minimum of fuss. Next, we see how to gain more control over the split and use the split-apply-combine pattern in PySpark.

SIDEBAR **Vocabulary matters: partitions vs. chunks**

It can be tempting to use the word *partitions* when talking about how PySpark splits the data for a pandas UDF. Spark already has a concept of partitions, though: they refer to the physical data contained on the worker nodes. When working with pandas UDF, Spark can use the partitions as chunks, but can also decide to split them or move some data around. Because of this, I use *chunks* or *groups* (for grouped pandas UDF) instead. Less confusion, more data fun.

8.3.4 Grouped map UDF

Grouped map UDF are PySpark's answer to the split-apply-combine pattern. At the core, split-apply-combine is just a series of three steps that are frequently used in data analysis.

1. First, you *split* your data set into logical chunks.
2. You then *apply* a function to each chunk independently.
3. Finally, you *combine* the chunks into a unified data set.

To be perfectly honest, I did not know this pattern's name until somebody pointed at my code one day and said "this is some nice split-apply-combine work you did there". You probably use it intuitively as well. In PySpark's world, I see it more as a *divide and process* move, as illustrated in figure 8.6.

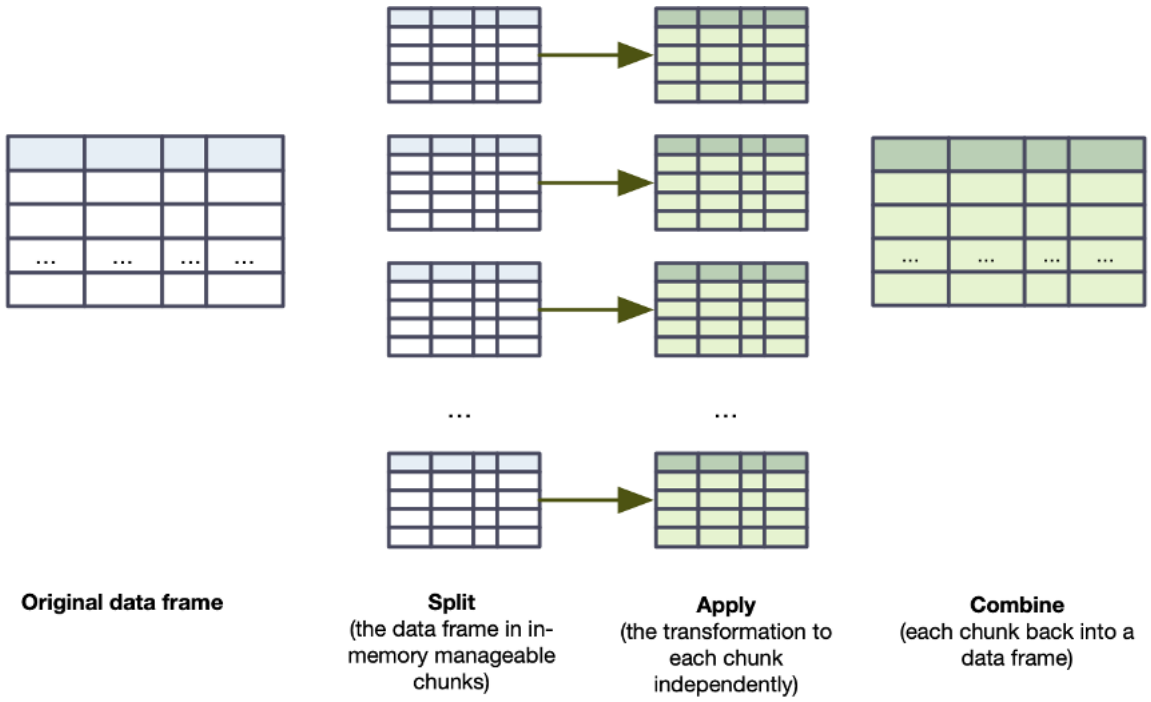


Figure 8.6 Split-apply-combine depicted visually. We chunk/group the data frame, process each one with pandas, before unioning them into a (Spark) data frame again.

Before looking at the PySpark plumbing, we focus on the pandas side of the equation. Where scalar UDF were relying on pandas Series, grouped map UDF are using pandas DataFrame. Each logical chunk from step 1 in figure 8.6 becomes a DataFrame ready for action. Our UDF must also return a DataFrame.

Grouped map UDF also take `pandas_udf()` as a decorator, this time with a type of `PandasUDFtype.GROUPED_MAP`. The return type is also more verbose: since we have multiple columns in the pandas DataFrame, we have to provide the schema in a `StructType()`. For a deeper dive into schemas, head to chapter 6.

Listing 8.16 A grouped map UDF that normalizes (min-max) the temperature given a set of values. The code in the `scale_temperature()` function is regular pandas code.

```
@F.pandas_udf(
    T.StructType(
        [
            T.StructField("stn", T.StringType()),
            T.StructField("year", T.StringType()),
            T.StructField("mo", T.StringType()),
            T.StructField("da", T.StringType()),
            T.StructField("temp", T.DoubleType()),
            T.StructField("temp_norm", T.DoubleType()),
        ]
    ),
    F.PandasUDFType.GROUPED_MAP,
)
def scale_temperature(temp_by_day):
    """Returns a simple normalization of the temperature for a site.

    If the temperature is constant for the whole window, defaults to 0.5."""
    temp = temp_by_day.temp
    answer = temp_by_day[["stn", "year", "mo", "da", "temp"]]
    if temp.min() == temp.max():
        return answer.assign(temp_norm=0.5)
    return answer.assign(temp_norm=(temp - temp.min()) / (temp.max() - temp.min()))
```

TIP `pandas_udf()` will take a SQL-like schema as a string parameter too. For listing 8.16, we could have used `stn string, name string, country string, year string, mo string, da string, temp double, temp_norm double` as the return type.

Compared with the UDF seen so far in this chapter, the main difference is the return type of the UDF. In both the Python and the scalar UDF, we returned a single column. Here, we return a *complete (pandas) DataFrame*. In listing 8.16, our return DataFrame contains six columns. My UDF only adds one new column, `temp_norm`, which scales the temperature column received from a scale from zero to one. Since I have a division in my UDF, I am giving a reasonable value of 0.5 if the minimum temperature in my chunk equals the maximum temperature. By default, pandas will give an infinite value for division by zero: PySpark will interpret this as null.

Now with the "apply" step done, the rest is a piece of cake. I broke the punch in chapter 5: we use `groupby()` to split a data frame in manageable chunks and then pass our function to the `apply()` method. You can see the result in listing 8.17.

Listing 8.17 Split-apply-combing in PySpark: we `groupby()` records in a `GroupedData` object and `apply()` our UDF to each group.

```
gsod = gsod.where(F.col("year") == "2018")  ❶
gsod = gsod.groupby("stn", "year", "mo").apply(scale_temperature)

gsod.show(5, False)
# +------+----+---+---+------------------+------------------+
# |stn   |year|mo |da |temp_c            |temp_norm         |
# +------+----+---+---+------------------+------------------+
# |010250|2018|12 |08 |-5.666666666666667 |0.06282722513088991|
# |010250|2018|12 |27 |-2.0555555555555554|0.40314136125654443|
# |010250|2018|12 |31 |-1.6111111111111103|0.4450261780104712 |
# |010250|2018|12 |19 |-2.4444444444444438|0.3664921465968586 |
# |010250|2018|12 |04 |2.5555555555555562 |0.8376963350785341 |
# +------+----+---+---+------------------+------------------+
# only showing top 5 rows
```

❶ If you are working locally, keeping a single year worth of data will make sure you don't wait too long to get your results.

I group by three fields, `stn`, `year`, and `mo`. Unlike the `groupby()/agg()` combo seen in chapter 5, where the keys are implicitly present in the resulting data frame, the UDF applied needs to return any column that we want in our resulting data frame. My UDF has six columns in its return value, the data frame after `apply()` has the same six, following the same type equivalence seen in table 8.1. In practice, you create a grouped map UDF with a `groupby()` pattern in mind, so there is a low risk of a mismatch.

WARNING With great power comes great responsibility: when grouping by your data frame, make sure each chunk is "pandas-size", i.e. it can be loaded comfortably in memory. If one or more chunks is too big, you'll get an out-of-memory exception.

Grouped map UDF shine when you have distinct groups of data that you can process independently. In the case of listing 8.17, we scale the temperature for each combination of `(station, year, month)`. The moment you feel like your code can process some distinct groups in your data frame, a grouped map UDF is a great choice.

8.3.5 Grouped aggregate UDF

We finish our tour of pandas user-defined functions with the grouped aggregate UDF. They can be thought of a combination of the ones we saw so far, as they take pandas Series as parameters but return a simple scalar value. In that sense, they are akin to Spark aggregate functions: each group is summarized by a single value.

For grouped aggregate UDF, we still rely on the "split" step provided by `groupby()` — which makes them like grouped map UDF — but this time, we apply our UDF to the `agg()` method,

just aggregate functions. For my grouped aggregate UDF, I wanted to do something a little more complex than reproducing the common offenders (count, min, max, average). In listing 8.18, I compute the linear slope of the (scaled) temperature for a given period, using scikit-learn's `LinearRegression` object. You do not need to know scikit-learn or machine learning to follow along: I'm using basic functionality and explain each step.

> **NOTE** This is not a machine learning exercise: I am just using scikit-learn's plumbing to create a feature. Machine learning in Spark is covered in part 3 of this book. Don't take this code as a robust model training exercise!

Listing 8.18 Creating a grouped aggregate UDF that computes the slope of a set of temperature

```
from sklearn.linear_model import LinearRegression  ❶


@F.pandas_udf(T.DoubleType(), F.PandasUDFType.GROUPED_AGG)
def rate_of_change_temperature(day, temp):
    """Returns the slope of the daily temperature for a given period of time."""
    return (
        LinearRegression()  ❷
        .fit(X=day.astype("int").values.reshape(-1, 1), y=temp)  ❸
        .coef_[0]  ❹
    )
```

❶ I import the linear regression object from `sklearn.linear_model`

❷ I initialize the `LinearRegression` object.

❸ The `fit` method trains the model, using the `day` Series as a feature and the `temp` series as the prediction.

❹ Since I have only one feature, I select the first value of the `coef_` attribute as my slope.

To train a model in scikit-learn, we start by initializing the model object. In this case, I use `LinearRegression()` without any other parameters. I then `fit` the model, providing `X`, my feature matrix, and `y`, my prediction vector. In this case, since I have a single feature, I need to `reshape` my `X` matrix or scikit-learn will complain about a shape mismatch.

At the end of the fit method, our `LinearRegression` object has trained a model and, in the case of a linear regression, keeps its coefficient in a `coef_` vector. Since I really just care about the coefficient, I just extract and return it.

It's easy to apply a grouped aggregate UDF to our data frame. In listing 8.19, I `groupby()` the station code, name, and country, as well as the year and the month. I pass my newly created grouped aggregate function as a parameter to `agg()`, passing my `Column` objects as parameter to the UDF.

Listing 8.19 Applying our grouped aggregate UDF using `agg()`. Our UDF behaves just like a Spark aggregate function.

```
result = gsod.groupby("stn", "year", "mo").agg(
    rate_of_change_temperature(gsod["da"], gsod["temp_norm"]).alias(  ❶
        "rt_chg_temp"
    )
)

result.show(5, False)
# +------+----+---+---------------------+
# |stn   |year|mo |rt_chg_temp          |
# +------+----+---+---------------------+
# |010250|2018|12 |-0.01014397905759162 |
# |011120|2018|11 |-0.01704736746691528 |
# |011150|2018|10 |-0.013510329829648423|
# |011510|2018|03 |0.020159116598556657 |
# |011800|2018|06 |0.012645501680677372 |
# +------+----+---+---------------------+
# only showing top 5 rows

result.groupby("stn").agg(
    F.sum(F.when(F.col("rt_chg_temp") > 0, 1).otherwise(0)).alias("temp_increasing"),
    F.count("rt_chg_temp").alias("count"),
).where(F.col("count") > 6).select(
    F.col("stn"),
    (F.col("temp_increasing") / F.col("count")).alias("temp_increasing_ratio"),
).orderBy(
    "temp_increasing_ratio"
).show(
    5, False
)
# +------+---------------------+  ❷
# |stn   |temp_increasing_ratio|
# +------+---------------------+
# |681115|0.0                  |
# |384572|0.0                  |
# |682720|0.0                  |
# |672310|0.0                  |
# |654530|0.08333333333333333  |
# +------+---------------------+
# only showing top 5 rows
```

❶ Applying a grouped aggregate UDF is the same as using a Spark aggregating function: you add it as an argument to the `agg()` method of the `GroupedData` object.

❷ I am looking at the stations having the lowest proportion of increasing temperature for a month, given that they have data for at least 6 distinct months. That can suggest data quality issues or skew in the data collection.

8.3.6 Going local to troubleshoot pandas UDF

Pandas UDF are quite useful to extend PySpark with transformations that are not included in the `pyspark.sql` module. I find that they're also quite easy to understand but pretty hard to get *right*. I finish this chapter with a few pointers on testing out and debugging your pandas UDF.

The most important aspect of an pandas UDF (and any UDF) is that it need to work on the

non-distributed version of your data. For regular UDF, this means passing *any argument of the type of values you expect* should yield an answer. As an example, our function in 8.2 took an array of two integers: it needs to work for any arrays of two integers, including a potential zero as a denominator. The same is true for any pandas UDF: you need to be lenient with the input you accept and strict with the output you provide.

To test your pandas UDF, my favorite strategy is always to bring a sample of the data locally (one chunk) and test my function. This way, I can play around in the REPL until I get it just right, then promote it to my script. I show an example of the `rate_of_change_temperature()` UDF, applied locally, in listing 8.20.

Listing 8.20 Moving one station, one month worth of data in a local pandas DataFrame to test my `rate_of_change_temperature()` function.

```
gsod_local = gsod.where("year = '2018' and mo = '08' and stn = '710920'").toPandas()


print(rate_of_change_temperature.func(gsod_local["da"], gsod_local["temp_norm"]))
# -0.007830974115511494
```

When bringing a sample of your data frame into a pandas data frame for a grouped map or grouped aggregate UDF, you need to ensure you're getting a full chunk to reproduce the results. In our specific case, since we grouped by `"station", "year", "month"`, I brought one station, one month (one specific year/month, to be precise) of data. Since the grouping of the data happens at PySpark's level (via `groupby()`), you need to think the filters for your sample data in the same fashion.

This is a very quick overview of a basic strategy to confirm your code is doing what you're expecting. In Chapter 14, I cover PySpark code testing, which includes testing UDF, both PySpark and pandas.

User-defined functions are probably the most powerful feature PySpark offers for data manipulation. While the standard data manipulation API provided a lot of functionality out of the box, you have the option to outgrow what's provided and write your own functions, using Python and pandas. Once written, scaling them to PySpark is as easy as decorating them. Python, pandas, Spark, they all work together now.

8.4 Summary

- The most low level and flexible way of running Python code within the distributed Spark environment is to use the resilient distributed dataset (RDD). With an RDD, you have no structure imposed on your data and need to manage type information into your program, and defensively code against potential exceptions.
- The API for data processing on the RDD is heavily inspired by the MapReduce framework. The same ideas are permeating to the data frame abstraction, which can be seen as a specialized and structured RDD.
- The data frame's most basic Python code promotion functionality, called the (PySpark) UDF, emulates the "map" part of the RDD. You use it as a scalar function, taking `Column` objects as parameters and returning a single `Column`.
- PySpark provides three UDF that leverages pandas serialization and processing (hence their name, "pandas UDF"): the scalar version, which provides similar functionality to the Python UDF, the grouped map, that splits the data frame into chunks and processes them using pandas code on a DataFrame, and the grouped aggregate, which takes `Columns` and processes them like pandas Series, returning a scalar value.

8.5 Exercises

8.5.1 Exercise 8.1

Using the following definitions, create a `temp_to_temp(value, from, to)` that takes a numerical `value` in `from` degrees and converts it to `to` degrees.

- `C = (F - 32) * 5 / 9` (Celcius)
- `K = C + 273.15` (Kelvin)
- `R = F + 459.67` (Rankine)

8.5.2 Exercise 8.2

Correct the following UDF so it doesn't generate an error.

```
@F.udf(T.IntegerType())
def naive_udf(t: str) -> str:
    ...
    return answer * 3.14159
```

8.5.3 Exercise 8.3

Modify listing 8.16 to use Celcius degrees instead of Fahrenheit. How is the result of the UDF different if applied to the same data frame?

8.5.4 Exercise 8.4

Taking listing 8.17, what happens if we apply our grouped map UDF like so instead?

```
gsod_exo = gsod.groupby("year", "mo").apply(scale_temperature)
```

8.5.5 Exercise 8.5

Modify listing 8.18 to return both the intercept of the linear regression, as well as the slope, in an `ArrayType`. (Hint: the intercept is in the `intercept_` attribute of the fitted model.)

10 A foray into machine learning: logistic regression with PySpark

This chapter covers

- how investing in a solid data manipulation foundation makes data preparation a breeze;
- how to address big data quality problems with PySpark;
- how to create custom features for your ML model;
- how to select features for your model depending on your use-case;
- how to assemble the data to get it ready for model training;
- how to train and evaluate your ML model.

I get excited doing machine learning, but not for the reasons most people are.

I love getting into a new data set and trying to solve a problem. Each data set sports its own problems and idiosyncrasies and I feel that getting it "ML-ready" is extremely satisfying. Building a model gives purpose to data transformation: you ingest, clean, profile, torture the data for a higher purpose: solving a real-life problem. This chapter takes a "clean-ish" data set and gets it all the way to modeling, leveraging a new corner of PySpark we've yet to explore. Just like in a real model, we will spend some time cleaning (and complaining) at our data before building a (pretty decent) first model. The exercises in this chapter will be a little different than what we've seen so far in this book: because PySpark provides a very coherent ML API I'll take advantage of this to let you try different options. I encourage you to crack open the API documentation and try your hand at them. The answer key has your back in any case!

SIDEBAR **This is not a masterclass in machine learning**

This chapter assumes a little familiarity with machine learning. I explain the concepts as I go along, but I can't cover the full modeling process as it would be a book on its own. If you are interested in building your own machine learning models, I strongly recommend *Introduction to Statistical Learning* by Gareth James, Daniela Witten, Trevor Hastie and Robert Tibshirani (Springer, 2013, freely available online at http://faculty.marshall.usc.edu/gareth-james/ISL/data.html). It uses R, but the concepts transcend languages. For a more practical (and Python-based) introduction, I really enjoyed *Real-World Machine Learning* by Henrik Brink, Joseph W. Richards, and Mark Fetherolf (Manning, 2016).

10.1 Reading, exploring and preparing our machine learning data set

This section covers the ingestion and exploration of our machine learning data set. More specifically, we'll review the content of our data frame, look at incoherences, and prepare our data for feature engineering.

For this chapter, we use a data set from Kaggle containing over 20,000 recipes, along with some high-level nutrition facts and many categories. You can get this data set online (https://www.kaggle.com/hugodarwood/epirecipes); I also included the data in the book's companion repository (under `data/recipes/epi_r.csv`). We start our program by setting our `SparkSession`: I am allocating 8 Gibibyte of RAM to my driver (see chapter 9 for more information about how Spark allocates memory). I also read the CSV data in listing 10.2. The code echoes the ingestion steps introduced in chapter 4; you can refer to it for a more in-depth discussion of CSV ingestion. I also print the dimensions of the data frame: 20,057 rows and 680 columns. I'll keep track of those dimensions as I clean the data frame.

Listing 10.1 Starting our `SparkSession` for our machine learning program.

```
 language="python" linenumbering="unnumbered">from pyspark.sql import SparkSession
import pyspark.sql.functions as F
import pyspark.sql.types as T

spark = (
    SparkSession.builder.appName("Recipes ML model - Are you a dessert?")
```

```
    .config("spark.driver.memory", "8g")
    .getOrCreate()
)
```

Listing 10.2 Ingesting our data set, printing the dimension and the schema

```
 language="python" linenumbering="unnumbered">food = spark.read.csv(
    "../../data/recipes/epi_r.csv", inferSchema=True, header=True
)

print(food.count(), len(food.columns))
# 20057 680

food.printSchema()
# root
#  |-- title: string (nullable = true)
#  |-- rating: string (nullable = true)
#  |-- calories: string (nullable = true)
#  |-- protein: double (nullable = true)
#  |-- fat: double (nullable = true)
#  |-- sodium: double (nullable = true)
#  |-- #cakeweek: double (nullable = true)
#  |-- #wasteless: double (nullable = true)
#  |-- 22-minute meals: double (nullable = true)
#  |-- 3-ingredient recipes: double (nullable = true)
#  |-- 30 days of groceries: double (nullable = true)
#  ...
#  |-- créme de cacao: double (nullable = true)
#  |-- crêpe: double (nullable = true)
#  |-- crme de cacao: double (nullable = true)
# ... and many more columns
```

1. Our data set starts with 20,057 rows and 680 columns.
2. Some of the columns contains undesirable characters, like a #…
3. Or a space
4. Or some invalid characters!

Just by looking at the schema, we can already see some not-so-desirable column names. I usually like my columns to all be lowercase, with underscores _ between the words. To do so, I create a python function `sanitize_column_name` that will take a "dirty" column name and return it clean. The function is then applied to all the columns in my data frame in one fell swoop, using the `toDF()` method, introduced in chapter 4. `toDF()`, when used to rename the columns of a data frame, takes as parameters N strings, where N is the number of columns in our data frame. Since we can access the columns of our

data frame via `food.columns`, a quick list comprehension takes care of renaming everything. I also unpack my list into distinct attributes using the star operator (see Appendix D for more details). Having a consistent column naming scheme will make subsequent code easier to write, read, and maintain in the long run: I treat column names like variables in a regular program.

Listing 10.3 Sanitizing my columns in one operation, using a Python function and the `toDF()` method.

```
language="python" linenumbering="unnumbered">def sanitize_column_name(name):
    """Drops unwanted characters from the column name.

    We replace spaces, dashes and slashes with underscore,
    and only keep alphanumeric characters."""
    answer = name
    for i, j in ((" ", "_"), ("-", "_"), ("/", "_"), ("&", "and")):
        answer = answer.replace(i, j)
    return "".join(
        [
            char
            for char in answer
            if char.isalpha() or char.isdigit() or char == "_"
        ]
    )


food = food.toDF(*[sanitize_column_name(name) for name in food.columns])
```

❶ I iterate over the characters I want to get rid of, replacing them by something more consistent.
❷ We only keep letters, numbers and underscore.

With this out of the way, we can now start exploring the data. In this section, we ingested and cleaned the column names of our data, making the data frame friendlier to work with. In the next section, we'll classify our columns as different kinds of features, assess the quality of our data, and fill the gaps.

10.1.1 Exploring our data and getting our first feature columns

This section covers digging into our data and encoding our first machine learning features. I introduce the main kind of machine learning features and how to easily keep track of the features that we feed into our model training.

Exploring data for machine learning is a lot similar to exploring data when performing a transformation in the sense that we manipulate the data to uncover some inconsistencies, patterns, or gaps. Because of this, all the material seen

during the previous chapters applies here. Talk about convenience! On the other hand, machine learning has a few idiosyncrasies that impact how we reason about and prepare data. In listing 10.4, I print a summary table for each of the columns in our `food` data frame. This takes a while but gives us a decent summary of the data contained in each column. Unlike single-node data processing, PySpark cannot necessarily assume that a column will fit into memory, so we can't go crazy with charts and extensive data profiling tools (see the tip below).

Listing 10.4 Creating a summary table of all our columns. I printed one of the 680.

```
 language="python" linenumbering="unnumbered">for x in food.columns:
    food.select(x).summary().show()

# many tables looking like this one.
# +-------+--------------------+
# |summary|               clove|
# +-------+--------------------+
# |  count|               20052|   ❶
# |   mean|0.009624975064831438|
# | stddev| 0.09763611178399834|
# |    min|                 0.0|   ❷
# |    25%|                 0.0|   ❷
# |    50%|                 0.0|   ❷
# |    75%|                 0.0|   ❷
# |    max|                 1.0|   ❷
# +-------+--------------------+
#
```

❶ The clove column contains 20,052 non-null values (so 5 records are null).
❷ Since the quartile distribution is only 0.0 and 1.0, there is a very good chance that our column is binary (0, 1).

TIP One of the best tips when processing data in PySpark is recognizing that your data is small enough to be gathered to a single node. For pandas data frames, the excellent pandas-profiling library can be used to automate a lot of the data profiling if your data is pandas-size (https://github.com/pandas-profiling/pandas-profiling). Remember: your Python knowledge doesn't go away when you use PySpark!

While looking at our summary data, we are looking at our numerical columns. In machine learning, we classify numerical features into two categories: *categorical* or *continuous*. A categorical feature is when your column takes a

discrete number, such as the month of the year (1 to 12). A continuous feature is when the column can be an infinity of possibility, such as the price of an item. We can subdivide the categorical family into three main types:

1. *Binary* (or *dichotomous*), when you have only two choices (0/1, True/False)
2. *Ordinal*, when the categories have a certain ordering (like the position in a race) that matters.
3. *Nominal*, when the categories have no specific ordering (like the color of an item).

Identifying your variables as categorical (with the proper sub-type) or continuous has a direct impact on the data preparation and, down the road, the performance of your machine learning model. Proper identification is dependent on the context (what does the column mean?) and how you want to encode its meaning. You'll develop a stronger intuition as you develop more ML programs. Don't worry if you don't get it right the first time: you can always come back and touch up your feature types. In chapter 11, we introduce ML pipelines which provide a nice abstraction for feature preparation, making it easy to evolve over time.

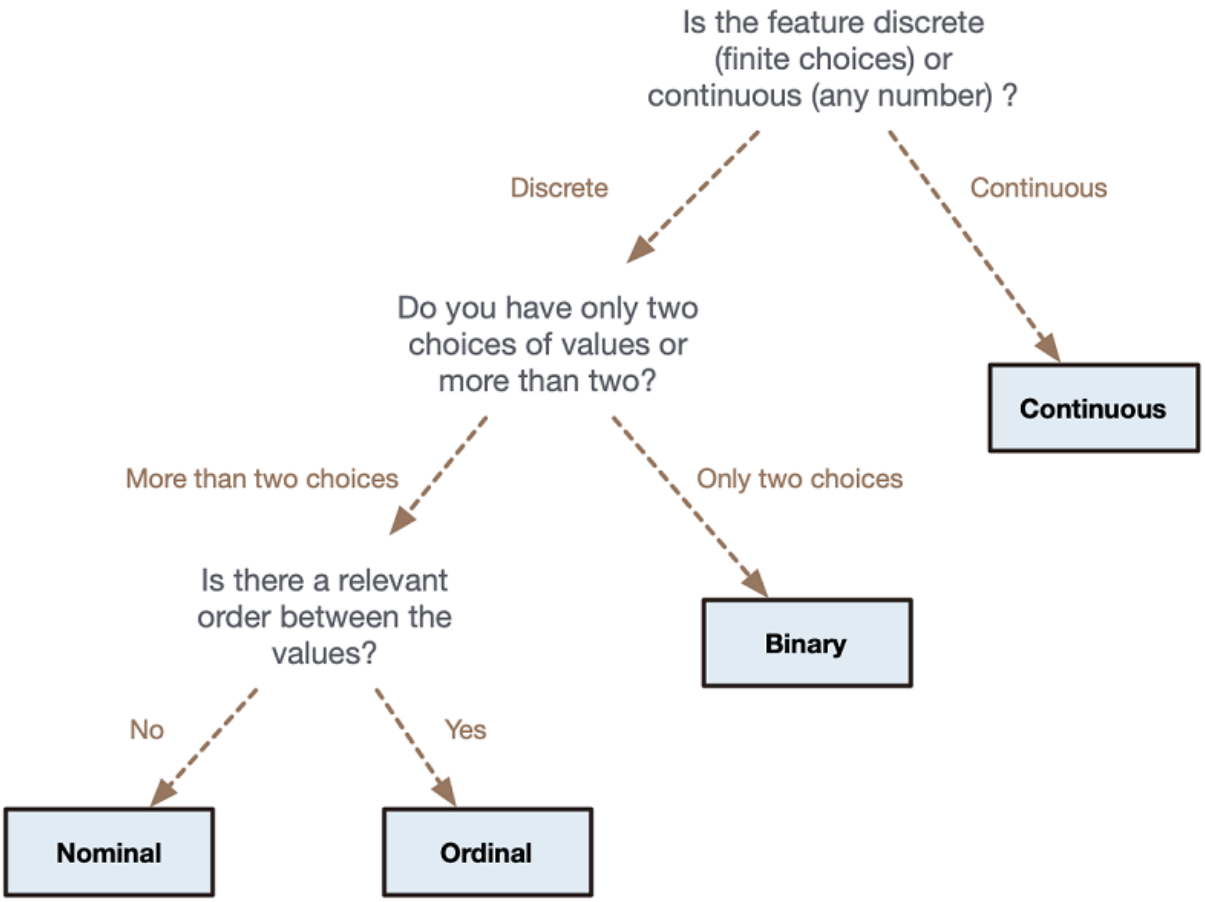


Figure 10.1 The different types of numerical features in a decision tree. Answer the questions to get which feature you are!

Looking at our summary data, it seems that we have a lot of potentially binary columns: in the case of the `clove` column, the minimum and three quartile values are all zero. To verify this, we'll group the entire data frame and collect a set of distinct values. If we have only two values for a given column, binary it is! In listing 10.5, I create a temporary data frame `is_binary` to identify the binary columns. I collect the results into a pandas data frame—since I know I have one row—and unpivot the result using the `unstack()` method available through pandas (PySpark has no easy way to unpivot). Most columns are binary, with the exception of the eight first. I am personally not convinced with `cakeweek` and `wasteless`: their name appears to me like they'd be yes/no questions. Time to investigate!

Listing 10.5 Identifying the binary columns from our data frame by computing the number of unique values.

```
 language="python" linenumbering="unnumbered">import pandas as pd

pd.set_option("display.max_rows", 1000)    ❶

is_binary = food.agg(
    *[
        (F.size(F.collect_set(x)) == 2).alias(x)    ❷
        for x in food.columns
    ]
).toPandas()

is_binary.unstack()    ❸

# title                    0    False
# rating                   0    False
# calories                 0    False
# protein                  0    False
# fat                      0    False
# sodium                   0    False
# cakeweek                 0    False
# wasteless                0    False
# 22_minute_meals          0     True
# 3_ingredient_recipes     0     True
# ... the rest are all = True
```

❶ pandas will display a handful of rows at a time. Setting this option will print at most 1,000 rows, which is helpful when exploring data.

❷ collect_set() will create a set of the distinct values as an array, and size() returns the length of the array. 2 distinct values means that it's probably binary.
❸ unstack unpivots a pandas data frame, making a wide data frame easier to analyze in the terminal.

This section covered the main types of numerical features we encounter in machine learning and identified the binary features in our data frame. In the next section, we'll perform some analysis on the remaining columns and establish our base set of features.

10.1.2 Addressing data mishaps and building our first feature set

In this section, we investigate some seemingly incoherent features and clean our data set following our findings. We also identify our first feature set along with each feature type.

At the end of the last section, we concluded that the vast majority of our feature columns in our data set were binary. Furthermore, there were two columns that were suspicious: `cakeweek` and `wasteless`. In listing 10.6, I display the discrete values both columns can take and then show the records where one of them contains a non-binary value.

Listing 10.6 Identifying the distinct values for our 2 suspicious columns as well as the offending records.

```
 language="python" linenumbering="unnumbered">food.agg(*[F.collect_set(x) for x in (
    1, False
)

# +------------------------------+----------------------+
# |collect_set(cakeweek)         |collect_set(wasteless)|
# +------------------------------+----------------------+
# |[0.0, 1.0, 1188.0, 24.0, 880.0]|[0.0, 1.0, 1439.0]    |
# +------------------------------+----------------------+

food.where("cakeweek > 1.0 or wasteless > 1.0").select(
    "title", "rating", "wasteless", "cakeweek", food.columns[-1]    ❶
).show()

# +--------------------+--------------------+---------+--------+------+
# |               title|              rating|wasteless|cakeweek|turkey|
# +--------------------+--------------------+---------+--------+------+
# |"Beet Ravioli wit...| Aged Balsamic Vi...|      0.0|   880.0|   0.0|
# |"Seafood ""Catapl...|            Vermouth|   1439.0|    24.0|   0.0|
# |"""Pot Roast"" of...| Aunt Gloria-Style "|      0.0|  1188.0|   0.0|
# +--------------------+--------------------+---------+--------+------+
```

❶ I print the first few records and the last to see potential data alignment problems.

Once more: the reason why I do not like CSV! For three records, it seems like our data set had a bunch of quotation marks along with some commas that confused PySpark's otherwise robust parser. In our case, since we have a small number of records affected, I did not bother with realigning the data and deleted them outright. I keep the null values as well.

Listing 10.7 Keeping only the legit values for `cakeweek` and `wasteless`.

```
 language="python" linenumbering="unnumbered">food = food.where(
    (
        F.col("cakeweek").isin([0.0, 1.0])    ❶
        | F.col("cakeweek").isNull()    ❶
    )
    & (
        F.col("wasteless").isin([0.0, 1.0])    ❶
        | F.col("wasteless").isNull()    ❶
    )
)

print(food.count(), len(food.columns))

# 20054 680    ❷
```

❶ This reads "if cakeweek AND wasteless are both either 0.0, 1.0 or null."
❷ Lost three records, as expected.

In listing 10.7, I sanity check my filtering by printing the dimensions of my data frame. It's not a perfect way to know if my code is bug-free but it helps to validate if I'm removing the right number of records. If something goes wrong-data wise, this information can help to pinpoint where in the code our data went wrong.

Now that we have identified two binary-in-hiding feature columns, we can identify our feature set and our target variable. The *target* (or *label*) is the column containing the value we want to predict. In our case, the column is aptly named `dessert`. In listing 10.8, I create ALL CAPS variables (to find them quickly) containing the four main sets of columns I'll care about:

1. The *identifiers*, which are the column(s) that contain the information unique to each record.
2. The *target*, which are the column(s) (most often one) that contain the value we wish to predict.
3. The *continuous* columns, containing continuous features.

4. The *binary* columns, containing the binary features.

The data set does not seem to contain categorical variables.

Listing 10.8 Creating four top-level variables to easily refer to the features without hard-coding them.

```
 language="python" linenumbering="unnumbered">IDENTIFIERS = ["title"]

CONTINUOUS_COLUMNS = [
    "rating",
    "calories",
    "protein",
    "fat",
    "sodium",
]

TARGET_COLUMN = ["dessert"]    ❶

BINARY_COLUMNS = [
    x
    for x in food.columns
    if x not in CONTINUOUS_COLUMNS
    and x not in TARGET_COLUMN
    and x not in IDENTIFIERS
]
```

❶ Although I have only one target, I find it convenient to put it into a list to be consistent with the other VARIABLES.

I like to keep track of my features through variables instead of deleting them from my data frame. It removes some of the guesswork when you get the data ready for the model training—*which columns are features, again?*—and it serves as light-weight documentation when reading your code the next time. It's basic, but it serves our purpose well here.

In this section, we rapidly cleaned our data. In practice, this stage will take north of half your time when building an ML model. Fortunately, data cleaning is principled data manipulation, so you can leverage all the PySpark toolkit you've built so far. We also identified our features and their type and grouped them into lists, which makes it easier to reference in the next sections. In the next section, we'll take care of the null imputation and finish our data cleanup by looking at the continuous feature columns.

10.1.3 Getting our data set ready for assembly: null imputation and casting

This section covers the last few steps before assembling our data set for machine learning. I review our columns for null values, explore the continuous columns, and perform the last bit of cleaning on them.

SIDEBAR **Keep some lab notes!**

In this section, we are doing quite a bit of back and forth. I did my best to arrange the data cleaning portion of our program relatively ordered. In all honesty, when building the original script for this chapter, I re-arranged many sections as I profiled and examined the data. Your own attempt at cleaning the data would have certainly yielded a very different program.

Data preparation for machine learning is part art, part science. There is a big part of intuition and experience that comes into play; you will end up recognizing some data patterns and create a personal library of strategies to deal with them. Because of this, it's crucial to document your steps to make it easier on your future self (and your colleagues!) when you pull that code back. I keep a notebook by my side when cleaning data, and I make sure to collect my "lab" notes in a format that will be easy to share.

As a first step, we tackle the null values from a full-data set approach. This means:

1. removing the records where all the features are null, and
2. removing the records where the target is null.

Depending on your use case (and your available time), you could also manually impute the null features and targets. This is called (manual) *labeling* and can be very useful if time-consuming. In our case here, I'll take the easy way out and drop the records missing crucial values.

Listing 10.9 Removing the records that have only `null` values, or where the target column is `null`.

```
 language="python" linenumbering="unnumbered">food = food.dropna(
    how="all",
    subset=[x for x in food.columns if x not in IDENTIFIERS],    ❶
)
```

```
food = food.dropna(subset=TARGET_COLUMN)  ❶

print(food.count(), len(food.columns))
# 20049 680  ❷
```

❶ I can use the feature group variables instead of having to remember which column is which.
❷ We lost 5 records in the process (20,054 - 5 = 20,049)

As a second step, I impute a default value to all my binary columns. As a rule of thumb, `1` means `True` and `0` means `False`. In the context of a binary variable, an absence of value can be thought of being conceptually closer to false than true, so we'll default every binary feature column to `0.0`.

Listing 10.10 Setting a default value of `0.0` to every binary feature column in our data frame.

```
 language="python" linenumbering="unnumbered">food = food.fillna(0.0, subset=BINARY_

print(food.where(F.col(BINARY_COLUMNS[0]).isNull()).count())  # => 0
```

Our binary feature columns are now ready to roll! Let's now tackle the continuous variables. Looking back at our schema in listing 10.2, because of some data misalignment, PySpark inferred the type of the `rating` and `calories` column as a string where they should clearly have been numerical. In listing 10.11, I create a simple UDF that takes a string column and returns `True` if the value it a floating-point number (or a null! PySpark will allow null values in a `Double` column) and `False` otherwise. I am doing this more as an exploration rather than a bona fide cleaning step; since a string value in any of those two columns means that the data is misaligned, I will drop the record rather than trying to fix it.

The UDF looks rather complicated, but if we take it slowly, it's very simple. I return `True` right off the bat if the value is null. If I have a non-null value, I try to cast the value as a Python `float`. If it fails, `False` it is!

Listing 10.11 Looking at the non-numerical values in the `rating` and `calories` columns.

```
 language="python" linenumbering="unnumbered">from typing import Optional


@F.udf(T.BooleanType())
```

```
def is_a_number(value: Optional[str]) -> bool:
    if not value:
        return True
    try:
        _ = float(value)
    except ValueError:
        return False
    return True


food.where(~is_a_number(F.col("rating"))).select(
    *CONTINUOUS_COLUMNS
).show()

# +---------+------------+-------+----+------+
# |   rating|    calories|protein| fat|sodium|
# +---------+------------+-------+----+------+
# | Cucumber| and Lemon "|   3.75|null|  null|
# +---------+------------+-------+----+------+
```

1. The underscore means "perform the work, but I don't care about the result".
2. We have one last rogue record!

We have a single remaining rogue record (damn those pesky unaligned CSVs!), that I remove in listing 10.12, before confidently casting the columns as a double. Our continuous feature columns are now all numerical.

Listing 10.12 Casting the `rating` and `calories` columns into double, after removing the offending record.

```
 language="python" linenumbering="unnumbered">for column in ["rating", "calories"]:
    food = food.where(is_a_number(F.col(column)))
    food = food.withColumn(column, F.col(column).cast(T.DoubleType()))

print(food.count(), len(food.columns))

# 20048 680
```

1. One record lost!

The last step is to impute the null values. Unlike the binary feature columns, there is no obvious answer about an appropriate default value for continuous variables. We could impute a `0.0` but I think it's weird to have a recipe at zero calories. I'll take a slightly more involved route and impute the average of the column as a default value. This is a common strategy for null imputation: giving null records the average value of the column it belongs to does not change the

average of the column. Since we have no other knowledge about the distribution of the values in our data frame, it's a perfectly reasonable starting point.

Before doing so, I want to look at the actual values to remove any ridiculous values that would break the computation of the average. In listing 10.13 I repeat the summary table I displayed in rapid-fire in listing 10.4. We see right off the bat that some dishes are way over the top! I could filter out the records once more, but this time, I'll cap the values to the 99th percentile, avoiding extreme (and potentially wrong) values.

Listing 10.13 Looking at the values in our continuous feature columns

```
 language="python" linenumbering="unnumbered">food.select("rating", "calories", "pro
    "mean", "stddev", "min", "1%", "5%", "50%", "95%", "99%", "max",
).show()

# +-------+------------------+------------------+------------------+
# |summary|            rating|          calories|           protein|
# +-------+------------------+------------------+------------------+
# |   mean| 3.714460295291301|6324.0634571930705|100.17385283565179|
# | stddev|1.3409187660508959|359079.83696340164|3840.6809971287403|
# |    min|               0.0|               0.0|               0.0|
# |     1%|               0.0|              18.0|               0.0|
# |     5%|               0.0|              62.0|               0.0|
# |    50%|             4.375|             331.0|               8.0|
# |    95%|               5.0|            1318.0|              75.0|
# |    99%|               5.0|            3203.0|             173.0|
# |    max|               5.0|       3.0111218E7|          236489.0|
# +-------+------------------+------------------+------------------+

# +-------+-----------------+-----------------+
# |summary|              fat|           sodium|
# +-------+-----------------+-----------------+
# |   mean|346.9398083953107|6226.927244193346|
# | stddev|20458.04034412409|333349.5680370268|
# |    min|              0.0|              0.0|
# |     1%|              0.0|              1.0|
# |     5%|              0.0|              5.0|
# |    50%|             17.0|            294.0|
# |    95%|             85.0|           2050.0|
# |    99%|            207.0|           5661.0|
# |    max|        1722763.0|       2.767511E7|
# +-------+-----------------+-----------------+
```

In listing 10.14, I hard-code the `maximum` acceptable values for each column and then I apply those maximum iteratively to my food data frame. I then compute

the mean for each column, returning a dictionary with the column name as keys and the value to replace the `null` with as values. With this, our data set is looking decent for machine learning!

Listing 10.14 Imputing the average value for four continuous columns, using the pre-computed values.

```
 language="python" linenumbering="unnumbered">maximum = {
    "calories": 3203.0,    (1)
    "protein": 173.0,    (1)
    "fat": 207.0,    (1)
    "sodium": 5661.0,    (1)
}

for k, v in maximum.items():
    food = food.withColumn(k, F.least(F.col(k), F.lit(v)))


def compute_mean(df, include):
    return (
        df.agg(*(F.avg(c).alias(c) for c in include))    (1)
        .first()
        .asDict()
    )


computed_mean = compute_mean(
    food, ["calories", "protein", "fat", "sodium"]
)

food = food.fillna(computed_mean)
```

(1) I hard-code the values here to make sure my analysis is consistent across runs. If the data changes, I might want to re-check if the 99-th percentile is still a good measure before automating the imputation, but I am comfortable with those exact values for the time being.

NOTE There is no surefire procedure to take care of outliers or identify them in the first place. 5,661mg of sodium is still criminally high, but more realistic considering some outrageous recipes available in the wild. In this chapter, I won't come back to it, but this would be one instance where I'd leave some breadcrumb after having a full cycle done to tweak my approach.

In this section, we imputed null records globally on our data set. We also cleaned the categorical feature columns using a small UDF. In the next section,

we'll perform some feature engineering and select the variables that will make their way into our model.

10.2 Feature engineering and selection

This section covers two important steps of model building: feature engineering and selection. We prepare a few custom features, look at how to encode the data into a Vector format, and assemble our final data set.

We could potentially spend a lot of time crafting more and more sophisticated features. Since my goal is to provide an end-to-end model using PySpark, I proceed with a first-round at everything. More specifically, we look at:

1. filtering the binary features using the Jaccard similarity;
2. creating a few custom features using our continuous feature columns;
3. measuring correlation over continuous features;
4. scaling the continuous features so they are consistent with our binary ones; and
5. assembling the final data set in a `Vector` column, ready for model training.

It is by no mean the only way we could approach this, but those five steps give a good overview of what can be done in PySpark.

10.2.1 Weeding out the rare binary occurrence columns

In this section, I remove the columns that are not present enough in the data set to be considered a reliable predictor.

Rarely occurring features are an annoyance when building a model as the machine can pick up a signal that is just there by chance. For example, if you are flipping a fair coin and get "head" and use that as a feature to a model predicting the next flip, you might get a dummy model that will predict 100% "head". It'll work perfectly until you get a tail… In the same vein, you want to have enough representation for each feature that goes into your model.

For this model, I choose 10 to be my threshold. If you are a binary feature with less than 10 of `0.0` or `1.0`, then I do not want you in my model. In listing 10.15, I compute the sum of each binary column; this will give me the number of 1.0 since the sum of ones is equal to their count. If the count of ones/sum of a column is below 10 or above the number of records minus 10, I collect the column name to remove it.

Listing 10.15 Removing the binary features that happen too little of too often.

```
language="python" linenumbering="unnumbered">inst_sum_of_binary_columns = [
```

```
    F.sum(F.col(x)).alias(x) for x in BINARY_COLUMNS
]

sum_of_binary_columns = (
    food.select(*inst_sum_of_binary_columns).head().asDict()    ❶
)

num_rows = food.count()
too_rare_features = [
    k
    for k, v in sum_of_binary_columns.items()
    if v < 10 or v > (num_rows - 10)
]

len(too_rare_features)  # => 167

print(too_rare_features)
# ['cakeweek', 'wasteless', '30_days_of_groceries',
#  [...]
#  'yuca', 'cookbooks', 'leftovers']

BINARY_COLUMNS = list(set(BINARY_COLUMNS) - set(too_rare_features))    ❷
```

❶ Since a row is just like a Python dictionary, I can bring the row back to the driver and process it locally.
❷ Rather than deleting the columns from the data frame, I just remove them from my BINARY_COLUMNS list.

We have removed 167 features that are either too rare or too frequent. While this number seems high, some of the features are very precise for a data set with a few thousand recipes. When creating your own model, you will certainly want to play around with different values to see if some parameters are still too rare to provide a reliable prediction.

In this section, we removed rare binary features, reducing our feature space by 167 elements. In the next section, we look at creating custom features that might improve the predictive power of our model.

10.2.2 Creating custom features

In this section, we look at creating new features from the data we have at hand. Doing so can improve our model interpretability and predictive power. I show one example of feature preparation that places a few continuous features on the same scale as our binary ones.

Fundamentally, creating a custom feature in PySpark is nothing more than creating a new column, with a little more thought and notes on the side. Manual feature creation is one of the secret weapons of a data scientist: you can embed

business knowledge into highly custom features that can improve your model accuracy and interpretability. As an example here, we'll take the `protein` and `fat` columns representing the quantity (in grams) of protein and fat in the recipe, respectively. With the information in those two columns, I create two features representing the percentage of calories attributed to each macro-nutriment.

Listing 10.16 Creating new features that compute the percentage of calories attributable to protein and fat. Creating new custom features is often nothing more than creating a new column.

```
 language="python" linenumbering="unnumbered">food = food.withColumn(
    "protein_ratio", F.col("protein") * 4 / F.col("calories")    ❶
).withColumn(
    "fat_ratio", F.col("fat") * 9 / F.col("calories")
)    ❶

food = food.fillna(0.0, subset=["protein_ratio", "fat_ratio"])

CONTINUOUS_COLUMNS += ["protein_ratio", "fat_ratio"]    ❷
```

❶ There are 4 kcal per grams of protein, 9 kcal per grams of fat.
❷ I add the two columns in my set of continuous features.

By creating those two columns, I integrate new knowledge into my data. Before I provided the energy per gram of fat and protein, nothing in the data set would have provided this. The model could have drawn a relationship between the actual quantity of fat/proteins and the total calories count independently, but we're making this more obvious by allowing the model to have access to the ratio of protein/fat (and carbs: see the sidebar at the end of this section) directly.

Will this small piece of code improve our model? Before we get to modeling, we'll want to remove the correlation between our continuous variables and assemble all our features into a single clean entity. This section was extremely short, but keep this lesson close to you when building a model: you can embed new knowledge in your data set by creating custom features. In PySpark, creating new features is done simply by creating columns with the information you want: this means you can create simple or highly sophisticated features.

SIDEBAR **Why not do the same with carbs? Avoiding multicollinearity.**

Without going too deep in how food translates to emerge, I did not compute a carbs ratio as a custom feature. Beyond the fact that the total amount of carbs is not provided (and that carbs absorption is a little more complex), we have to consider about the *linear dependence* (or *multicollinearity*) of our variables when working with certain types of models.

A linear dependence between variables happens when you have one column that can be represented as a linear combination of others. You could have been tempted to approximate the ratio of calories coming from carbs using the following formula:

Total Calories = 4 * (g of carbs) + 4 * (g of proteins) + 9 * (g of fat)

Or, when using a ratio based approach:

1 = (% of calories from carbs) + (% of calories from proteins) + (% of calories from fat)

In both cases, we introduce a linear dependency: we (and the machine) can compute the values of a column using nothing but the values of other columns. When using a model that has a linear component, such as the linear regression and the logistic regression, this will cause problems with your model accuracy (either under-fitting or over-fitting).

Multicollinearity can happen even if you pay attention to your variable selection. For more information, I recommend referring to *Introduction to Statistical Learning* (Springer, 2013. See the beginning of the chapter for a free downloadable link), section 3.3.3.

10.2.3 Removing highly correlated features

In this section, we take our set of continuous variables and we look correlation between them, in order to improve our model accuracy and explainability. I explain how PySpark builds a correlation matrix, the `Vector` and `DenseMatrix` objects, and how we can extract data from those objects for decision-making.

Correlation in linear models is not always bad: as a matter of fact, you want your features to be correlated with your target (this provides predictive power). On the other hand, we want to avoid correlation between features for two main reasons:

1. If two features are highly correlated, it means that they provide almost the same information. In the context of machine learning, this can confuse the fitting algorithm and create model or numerical instability.
2. The more complex your model, the more complex the maintenance. Highly correlated features rarely provide improved accuracy (see #1), yet complicates the model. Simple is

better.

For computing correlation between variables, PySpark provides the `Correlation` object. `Correlation` has a single method, `corr`, that computes the correlation between features in a `Vector`. Vectors are like PySpark arrays but with a special representation optimized for ML work (see 10.2.5 for a more detailed introduction). In listing 10.17, I use the `VectorAssembler` transformer on the `food` data frame to create a new column `continuous_features` that contains a `Vector` of all our continuous features.

A transformer is a preconfigured object that, as its name indicated, transforms a data frame. Independently, they look like an unnecessary complexity, but they shine when applied within a pipeline. I cover ML Pipelines in chapter 11.

Listing 10.17 Using the `VectorAssembler` transformer to assemble all the continuous feature columns into a single `Vector` column.

```
 language="python" linenumbering="unnumbered">from pyspark.ml.feature import VectorA

continuous_features = VectorAssembler(
    inputCols=CONTINUOUS_COLUMNS, outputCol="continuous_features"
)

vector_variable = continuous_features.transform(food)

vector_variable.select("continuous_features").show(3, False)

# +-----------------------------------------------------------------------+
# |continuous_features                                                    |
# +-----------------------------------------------------------------------+
# |[2.5,426.0,30.0,7.0,559.0,0.28169014084507044,0.14788732394366197]     |
# |[4.375,403.0,18.0,23.0,1439.0,0.17866004962779156,0.5136476426799007]  |
# |[3.75,165.0,6.0,7.0,165.0,0.14545454545454545,0.38181818181818183]     |
# +-----------------------------------------------------------------------+
# only showing top 3 rows

vector_variable.select("continuous_features").printSchema()

# root
#  |-- continuous_features: vector (nullable = true)
```

NOTE	Correlation will not work well if you blend categorical and/or binary features together. Choosing the appropriate dependency measure depends on your model, your interpretability needs, and your data. Check out, for instance, the Jaccard distance measure for non-continuous data.

In listing 10.18, I apply the `Correlation.corr()` function on my continuous feature vector and export the correlation matrix into an easily interpretable pandas data frame. PySpark returns the correlation matrix in a `DenseMatrix` column type, which is like a two-dimensional vector. In order to extract the values in an easy to read format, we have to do a little method juggling.

1. We extract a single record as a list of `Row` using `head()`.
2. A `Row` is like an ordered dictionary, so we can access the first (and only) field containing our correlation matrix using list slicing.
3. A `DenseMatrix` can be converted into a pandas-compatible array by using the `toArray()` method on the matrix.
4. Finally, we can directly create a pandas DataFrame from our Numpy array. Inputting our column names as an index (in this case, they'll play the role "row names") and column names makes our correlation matrix is uber-readable.

Listing 10.18 Creating a correlation matrix in PySpark. The resulting `DenseMatrix` is directly compatible with a NumPy array, and extracting it in Pandas is the easiest way to interpret it.

```
 language="python" linenumbering="unnumbered">from pyspark.ml.stat import Correlatio

correlation = Correlation.corr(
    continuous_features.transform(food), "continuous_features"    ❶
)

correlation.printSchema()

# root
#  |-- pearson(binary_features): matrix (nullable = false)    ❷

correlation_array = correlation.head()[0].toArray()    ❸

correlation_pd = pd.DataFrame(
    correlation_array,    ❹
    index=CONTINUOUS_COLUMNS,    ❹
    columns=CONTINUOUS_COLUMNS,    ❹
)
```

```
print(correlation_pd.iloc[:, :4])

#                 rating  calories   protein       fat  ❺
# rating         1.000000 -0.019631 -0.020484 -0.027028  ❺
# calories      -0.019631  1.000000  0.958442  0.978012  ❺
# protein       -0.020484  0.958442  1.000000  0.947768  ❺
# fat           -0.027028  0.978012  0.947768  1.000000  ❺
# sodium        -0.032499  0.938167  0.936153  0.914338  ❺
# protein_ratio -0.026485  0.029879  0.121392  0.086444  ❺
# fat_ratio     -0.010696 -0.007470  0.000260  0.029411  ❺

print(correlation_pd.iloc[:, 4:])

#                 sodium  protein_ratio  fat_ratio
# rating        -0.032499      -0.026485  -0.010696
# calories       0.938167       0.029879  -0.007470
# protein        0.936153       0.121392   0.000260
# fat            0.914338       0.086444   0.029411
# sodium         1.000000       0.049268  -0.005783
# protein_ratio  0.049268       1.000000   0.111694
# fat_ratio     -0.005783       0.111694   1.000000
```

❶ The corr method takes a data frame and a Vector column reference as a parameter and generates a single-row, single-column data frame containing the correlation matrix.
❷ The resulting DenseMatrix (shown as matrix in the schema) is not easily accessible by itself.
❸ Since the data frame is small enough to be brought locally, we extract the first record with head(), the first column via an index slice and we export the matrix as a NumPy array via toArray().
❹ The easiest way to interpret the correlation matrix is to create a pandas data frame. We can pass out column names as both index and columns, for easy interpretability.
❺ The correlation matrix gives the correlation between each field of the vector. The diagonal is always 1.0 because each variable is perfectly correlated with itself.

When working with summary measures, such as the correlation of hypothesis tests, PySpark will often delegate the extraction of values to a simple NumPy or pandas conversion. Instead of remembering a different series of method-juggling for each scenario, I use the REPL and the inline documentation:

1. Look at the schema of your data frame and the documentation of the method/function used. `matrix` or `vector`? They are NumPy arrays in disguise.
2. Since your data frame will always fit in memory, bring the desired records and extract the structures using `head()`, `take()` and the methods available on `Row` objects.
3. Finally, either wrap your data in a pandas data frame, a list of your structure of choice.

Once again, our `CONTINUOUS_COLUMNS` variable avoided a ton of typing and potential errors. More love for keeping track of our features when manipulating our data frame!

The last step from our correlation computation is to assess which variables we want to keep and which we want to drop. There is no absolute threshold for keeping or removing correlated variables (nor is there a protocol for *which* variable to keep). From the correlation matrix in listing 10.18, we see high-correlation between sodium, calories, protein, and fat. Surprisingly, we see little correlation between our custom features and the columns that contributed to their creation. In my lab notes, I'd collect the following action items (left in exercises):

1. Explore the relationship between calorie count and the ratio of macro-nutriments (and sodium). Is there a pattern there or the calorie count (or size of portions) just all over the place.
2. If the calorie/protein/fat/sodium content related to the "dessertness" of the recipes? I can't imagine a dessert being very salty…
3. Run the model with all features, then with calories and protein removed. What is the impact on performance?

The correlation analysis raised more questions than it answered. This is a very good thing: data quality is a very complicated thing to assess. By keeping track of elements to explore in greater detail, we can move quickly to modeling and have a (crude) benchmark to anchor our next modeling cycles. We refactor our Python programs and ML models are no exception!

In this section, I covered how PySpark computes the correlation between variables and provides the results in a matrix. We saw the `Vector` and `Matrix` objects and how we can extract values from them. Finally, we assessed the correlation between our continuous variables and made a decision about their inclusion in our first model. In the next section, I cover variable scaling, an important but often overlooked component when building a linear model.

10.2.4 Scaling our features

This section covers variable scaling using the `MinMaxScaler` transformer. I also explain the rationale behind scaling variables.

Scaling variables means performing a mathematical transformation on the variables so they are all on the same numeric scale. When using a linear model, having scaled features means that your model coefficient (the weight of each feature) are comparable with one another. For example, if you have a variable that contains a number between 0.0 and 1.0 and another that contains a number

between 1,000.0 and 2,500.0, if they both have the same model coefficient (let's say 0.48), you might be tempted to think that both are equally important. In reality, the second variable is much more important because 0.48 times something in the thousands is much higher than 0.48 times something between zero and one.

To choose the right scaling algorithm, we need to look at our variables as a whole, since we have so many binary variables, it would be convenient to have every variable to be between zero and one. Furthermore, our `protein_ratio` and `fat_ratio` are ratios between zero and one too! PySpark provides for this exact use-case the `MinMaxScaler`: for each column, it divides each value by the *amplitude* of the column (or `max(column) - min(column)`). Easy peasy!

In listing 10.19, I create a `MinMaxScaler` estimator and provide the vector column input and output columns as strings. Estimators are very similar to transformers: in essence, an estimator needs to be *fitted* on the data, which yields a transformer that can transform the data. ML models are the quintessential estimators, but in the case of the `MinMaxScaler`, we fit the scaler on the data, computing the minimum and maximum values for each column forming the vector. If this does not make complete sense yet, estimators and transformers and much easier to understand in the context of pipelines, which chapter 11 is all about.

Listing 10.19 Scaling our non-scaled continuous variables, after assembling them into a vector.

```
 language="python" linenumbering="unnumbered">from pyspark.ml.feature import MinMaxS

CONTINUOUS_NB = [x for x in CONTINUOUS_COLUMNS if "ratio" not in x]

continuous_assembler = VectorAssembler(
    inputCols=CONTINUOUS_NB, outputCol="continuous"
)

food_features = continuous_assembler.transform(food)

continuous_scaler = MinMaxScaler(
    inputCol="continuous", outputCol="continuous_scaled",
)

food_features = continuous_scaler.fit(food_features).transform(
    food_features
)

food_features.select("continuous_scaled").show(3, False)
# +------------------------------------------------------...+
# |continuous_scaled                                     ...|
# +------------------------------------------------------...+
```

```
# |[0.5,0.13300031220730565,0.17341040462427745,0.0338164...|
# |[0.875,0.12581954417733376,0.10404624277456646,0.11111...|
# |[0.75,0.051514205432407124,0.03468208092485549,0.03381...|
# +-----------------------------------------------------...+
# only showing top 3 rows
```

TIP Check the `pyspark.ml.feature` module for other scalers. `StandardScaler`, which normalize your variables by subtracting the mean and then dividing by the standard deviation is also a favorite among data scientists.

All the variables in our `continuous_scaled` vector are now between zero and one. Our continuous variables are ready, our binary variables are ready; I think we are now to assemble our data set for machine learning! In this section, we reviewed the `MinMaxScaler` estimator and how we can scale variables so that they have the same amplitude. The next section will wrap all of our variables in a single vector, ready for modeling.

10.2.5 Assembling the final data set with the Vector column type

This section explores the vector data structure in the context of model preparation. We review the variables being inputted in the model and how to assemble them seamlessly using the `VectorAssembler`. I finally introduce the ML metadata PySpark imputes when assembling columns so we can easily remember what's where.

PySpark provides a special data structure when working with machine learning: the vector. In 10.2.3, I explained that a vector can be thought of as an array. More specifically, `Vectors` have two representations:

1. a *dense* representation, where a `Vector` in PySpark is simply a NumPy (single-dimensional) array object;
2. a *sparse* representation, where a `Vector` in PySpark is an optimized sparse vector compatible with the SciPy `scipy.sparse` matrix.

In practice, you don't decide if a `Vector` is sparse or dense: PySpark will convert between the two as needed. I bring the difference up since they *look different* when you `show()` them within a data frame. We already saw the dense vector representation (just like an array) in 10.2.3. To illustrate a sparse vector, I wrote the same vector twice, using the two different notations. A sparse vector is a triple containing

1. the length of a vector;
2. an array of positions where the elements are non-zero;
3. an array of non-zero values.

```
Dense:  [0.0, 1.0, 4.0, 0.0]
Sparse: (4, [1,2], [1.0, 4.0])
```

TIP In case you need them, `pyspark.sql.linalg.Vectors` has functions and methods for creating your vectors from scratch.

Sparse vectors (and matrices) are convenient when a lot of its values are zero; this happens quite often in machine learning. As an example, in listing 10.20, we use the `VectorAssembler` transformer once-more, but this time, we pass all the columns that we want to use in our model training. Since we have a ton of low-occurring binary features, the resulting `features` vectors are sparse.

Listing 10.20 Final assembly of our features into our `features` vector. We're getting there!

```
 language="python" linenumbering="unnumbered">model_binary_assembler = VectorAssembl
    inputCols=BINARY_COLUMNS
    + ["continuous_scaled"]
    + ["protein_ratio", "fat_ratio"],
    outputCol="features",
)

food_model = model_binary_assembler.transform(food_features)

food_model.select("title", "dessert", "features").show(5, truncate=30)
# +------------------------------+-------+------------------------------+
# |                         title|dessert|                      features|
# +------------------------------+-------+------------------------------+
# |Lentil, Apple, and Turkey W...|    0.0|(513,[21,36,106,146,177,189...|
# |Boudin Blanc Terrine with R...|    0.0|(513,[123,168,170,241,249,2...|
# | Potato and Fennel Soup Hodge |    0.0|(513,[21,95,157,180,214,235...|
# |Mahi-Mahi in Tomato Olive S...|    0.0|(513,[38,58,82,129,149,168,...|
# |     Spinach Noodle Casserole |    0.0|(513,[2,21,168,187,200,214,...|
# +------------------------------+-------+------------------------------+
# only showing top 5 rows
```

❶ We have 513 features getting fed into our model. The vector containing those features is sparse since most binary features are not happening often.

PySpark gladly took our `Vector` column `continuous_scaled` and

inserted all the values into the `features` vector. The ability to combine vectors using the `VectorAssembler` makes our features assembly much easier. Binary, categorical, and continuous features usually have a different treatment, and keeping them separate until the final assembly makes our code clearer and less prone to errors. In chapter 11, I revisit this property when building machine learning pipelines.

Now that everything is in a vector, how do we remember what's where? In chapter 6, I mentioned briefly that PySpark allows for a metadata dictionary to be attached to a column, and that this metadata is used when using PySpark's machine learning capabilities. Well now's the time! Let us peek at that metadata.

Listing 10.21 Getting PySpark to unfold the metadata, so we can remember what's where

```
 language="python" linenumbering="unnumbered">print(food_model.schema["features"])

# StructField(features,VectorUDT,true)


print(food_model.schema["features"].metadata)    ❶
# {'ml_attr':  {'attrs':  {'numeric': [
# {'idx': 0, 'name': 'fat_free'},
# {'idx': 1, 'name': 'advance_prep_required'},
# {'idx': 2, 'name': 'pasta'},
# [...]
# {'idx': 509, 'name': 'continuous_scaled_3'},    ❷
# {'idx': 510, 'name': 'continuous_scaled_4'},
# {'idx': 511, 'name': 'protein_ratio'},
# {'idx': 512, 'name': 'fat_ratio'}]}, 'num_attrs': 513}}
```

❶ The column schema for an assembled vector will keep track of the features making its composition under the metadata attribute.

❷ For scaled variables, since they originate from a VectorAssembler PySpark gives them a generic name, but you can track their name from the original VectorAssembler as needed.

Before going on to the modeling phase, let's have a quick summary of the work accomplished.

This section covered the assembly into a final feature vector, the last stage before sending our data for training. We revisited and explored the `Vector` data structure in greater detail, interpreting its dense and sparse representation. Finally, we used the `VectorAssembler` transformer to combine all our features, including those already in a `Vector`, and presented the metadata contained in the

`features` vector, facilitating the retrieval of individual features an otherwise anonymous vector. The next section covers the training and analysis of the model (finally!).

10.3 Training and evaluating our model

This section covers the training of our model. More specifically, I discuss the train/test split for our data set, re-introduce the evaluator in the concept of machine learning, and analyze the results. Finally, I explain how to save a fitted model for later re-use.

After making such pain-staking data preparation, cleaning, and assembly, I find the model training stage a little "meh". In the eyes of PySpark, a machine learning model is nothing more than another estimator: you `fit()` the estimator on your data, and then `transform()` a new data frame (containing a compatible `Vector` column) for predicting. This might be why so many people joke that machine learning is only knowing two methods: fit and transform!

For this section, I decided to fit a logistic regression, as mentioned in the chapter title. My decision is motivated by three main reasons.

First, I chose a *classification algorithm* because my target is finite (1.0 if the recipe is a dessert, 0.0 otherwise). On the flip side, a *regression algorithm* would have been apropos if I wanted to predict the number of calories (which can be any number and is not limited by any group, or classes). Logistic regression, despite having regression in its name, is used as a classification model (this is one of the most over-used interview questions, *hint hint*).

Second, logistic regression is part of the family of generalized linear models. Those models are well understood, quite powerful yet easier to explain than other models (like decision trees and neural networks). Despite its simplicity, logistic regression is omnipresent in a classification setting: most famous is the credit score, which to this day is powered by logistic regression models.

Finally, linear models are quite sensitive to data distribution, which gave me the perfect excuse for introducing scaling and variable correlation. Them being difficult made our data prep job a little more complex, which is perfect when learning the ropes!

Without further ado, here is our model training. Behold!

Listing 10.22 A logistic regression, in less than 10 lines of code (without counting data prep)

```
 language="python" linenumbering="unnumbered">from pyspark.ml.classification import

lr = LogisticRegression(
    featuresCol="features", labelCol="dessert", predictionCol="prediction"
)

train, test = food_model.randomSplit([0.7, 0.3])


train.cache()

lrModel = lr.fit(train)

predictions = lrModel.transform(test)
```

That's… it. In a few lines of code, we

1. Created a `LogisticRegression` estimator object, passing as parameters the column containing our *features* vector, our *label* (or target) column, as well as the desired prediction column.
2. Split our machine learning data set in a *train* and *test* subsets, randomly. This is to avoid over-fitting, where the model would predict very accurately on data it has seen (via training) and then tank when confronted with new data. The test set displays a more realistic view of how the model predicts on unknown data.
3. Fit our model estimator on the training data set.
4. Create a new data set `predictions` that contains our model prediction on the test set.

NOTE Remember in chapter 9 where I spoke about `cache()` being appropriate for ML training? This is it! Caching before model training will improve performance, but remember about spilling if your model is beyond the memory available.

With a model now trained and a prediction made on a testing set, we're ready for the last stage of our model: evaluating and dissecting our model. If you have not pulled a notebook yet, now's the time!

SIDEBAR **The logistic regression in a nutshell**

Feel free to skip this section if math is not your cup of tea.

It's easier to think about the logistic regression if we understand the linear model first. In school, you probably have learned about the simple one-variable regression displayed below. In this case, y is the dependent variable/target, x is the dependent variable/feature, and m is `x's coefficient, while `b is the *intercept* (or the value of the coefficient is zero).

```
y = m * x + b
```

A linear regression takes this simple formula and applies it to multiple features. In other words, `x` and `m` becomes vectors of value. If we use an index-based notation, it would look like this. (Some statistics textbook might use a different but equivalent notation).

```
y = b + (m0 * x0) + (m1 * x1) + (m2 * x2) + ... + (mn * xn)
```

Here, we have *n* features and coefficients. This linear regression formula is called the *linear component* and is usually written `X` (x for the observations, (beta) for the coefficients vector). A linear regression's prediction can span any number from negative infinity to infinity: there are no boundaries to the formula.

How do we get a classification model out of this? Behold, the logistic regression!

The logistic regression takes its name from the *logit* transformation. The logit transformation takes our linear component `X` and yields a function that is between zero and one. The formula below is the expanded form of the logistic function: note the location of the linear component. It looks like an arbitrary choice of function, but there is a lot of theory behind and this form is really convenient.

```
y = 1 / (1 + exp(-X))
```

The `y` of the logistic function will return for any value of `X` a number from zero to one. For turning this into a binary feature, we apply a simple threshold `1 if y >= 0.5 else 0`. If you want your model to be more or less sensitive, you can change this threshold. If you are curious about the raw `y` result of logistic regression, check the `rawPrediction` column of your prediction data set: you'll get a vector containing `[X, -X]`. The `probability` column will contain the `y` as defined in the logit formula, for both values in `rawPrediction`.

```
predictions.select("prediction", "rawPrediction", "probability").show(
    3, False
)

# +----------+----------------------+------------------------+
# |prediction|rawPrediction         |probability             |
# +----------+----------------------+------------------------+
# |1.0       |[-10.445268,10.445268]|[2.908472E-5,0.9999709] |
# |0.0       |[13.380082,-13.380082]|[0.9999984,1.5456218E-6]|
# |0.0       |[25.393660,-25.393660]|[0.9999999,9.368567E-12]|
# +----------+----------------------+------------------------+
```

10.3.1 Assessing model accuracy with the Evaluator object

This section covers the `Evaluator` object for binary classification and it's used for model assessment and comparison. We review a few metrics and compare our training and testing accuracy.

How well did we do for a first run? Even without an ounce of statistical judgment, we can think of two main metrics to pull out:

1. How accurate are we to predict desserts (`desserts = 1.0`)
2. How accurate are we to predict *not* desserts (`desserts = 0.0`)

While we can compute this easily using our data manipulation skills, PySpark provides functionality to perform those frequent and repetitive tasks. We look here at two metrics in particular:

1. the *confusion matrix*, which gives us a 2x2 matrix of predictions vs labels
2. the *receiver operating characteristic curve* (or ROC curve), which show the diagnostic ability of our model as we change its prediction threshold (more on that later).

For the confusion matrix, we use the `confusionMatrix()` method available in the `pyspark.mllib.evaluation.MulticlassMetrics` object. This object has not made its way into the `pyspark.ml`, so we are stuck using the RDD-based API. Fortunately for us, creating an RDD compatible with `MulticlassMetrics` could not be easier: we convert a two-column data frame (prediction as a first column, label as a second) into an RDD, and pass that as a parameter. In listing 10.23, I create the `MulticlassMetrics` object, initialize it with our now-RDD prediction data, and convert the `DenseMatrix` into a friendly pandas DataFrame.

Listing 10.23 Returning a confusion matrix from our testing data frame. The confusion matrix is a `DenseMatrix`.

```
 language="python" linenumbering="unnumbered">from pyspark.mllib.evaluation import M

# Create (prediction, label) pairs
predictionAndLabel = predictions.select("prediction", "dessert").rdd

metrics = MulticlassMetrics(predictionAndLabel)     ❶

confusion_matrix = pd.DataFrame(
    metrics.confusionMatrix().toArray(),
    index=["dessert=0", "dessert=1"],     ❷
    columns=["predicted=0", "predicted=1"],     ❷
)

print(confusion_matrix)

#            predicted=0  predicted=1
# dessert=0       4924.0         84.0
# dessert=1        102.0        987.0
```

❶ We pass the generated RDD containing (prediction, label) to the MultiClassMetrics class initializer.
❷ Just like with the confusion matrix, setting relevant axes to our matrix before printing makes the results easier to understand.

From this confusion matrix, we can compute many metrics. The two most important in a binary classification setting are the *precision* and *recall*. Both are easily explained in the context of the confusion matrix as I do in listing 10.24. PySpark will also give you the precision and recall directly through the `MulticlassMetrics` object, as well as the f1-score, a balanced compromise between precision and recall. Since the `MulticlassMetrics` works for multi-class classification too, we need to provide the label we consider "good": in our case, we have only one, `1.0`.

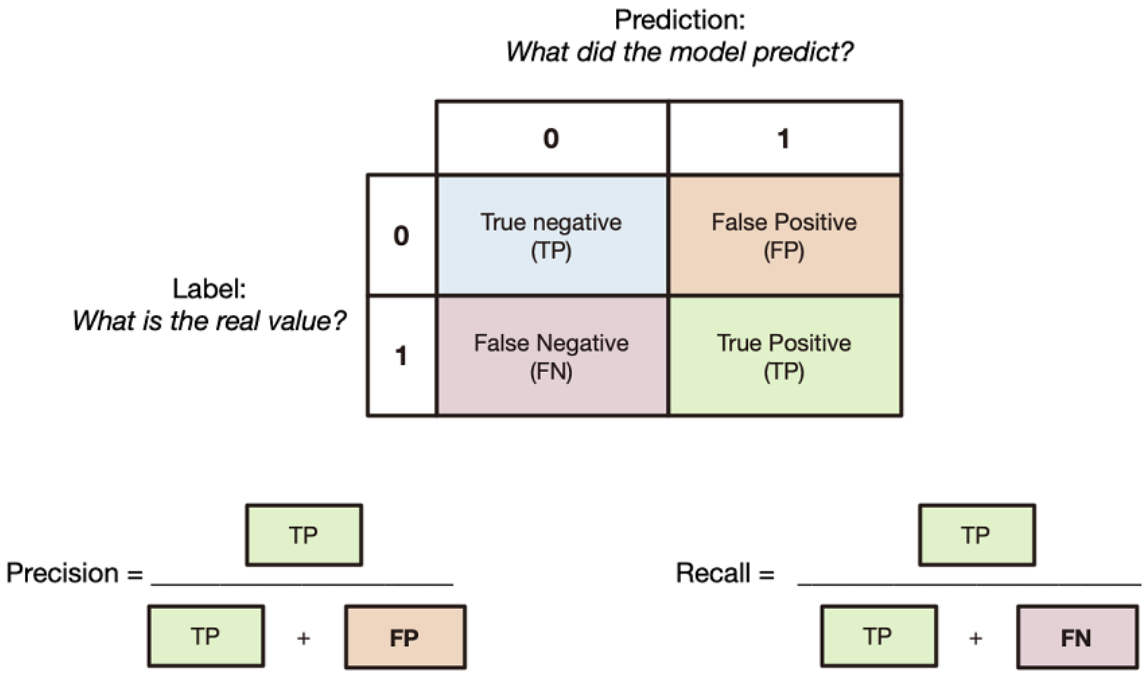


Figure 10.2 A visual depiction of precision and recall, as seen in a confusion matrix.

Listing 10.24 Listing the precision and recall directly from the `MulticlassMetrics` object.

```
 language="python" linenumbering="unnumbered">print(f"Model precision: {metrics.prec
print(f"Model recall: {metrics.recall(1.0)}")
print(f"Model f1-score: {metrics.fMeasure(1.0)}")

# Model precision: 0.9215686274509803
# Model recall: 0.90633608815427
# Model f1-score: 0.913888888888889
```

This is pretty good! Given that all three have a theoretical limit of 1 (or 100%), having precision and recall score of over 90% right off the bat is really encouraging. Now, we look at the ROC curve. Binary models can have their "sensitivity" tweaked, meaning that you can make a model predict `1.0` more often by changing the threshold used internally. (See the *"The logistic regression in a nutshell"* sidebar for more information) The ROC plays double duty by giving us a sense of precision and recall across multiply sensitivities. It allows us to optimize our decision-making (regarding precision vs. recall) and also gives us a sense of how robust is the model in general. Rather than explaining the metric and then showing it, let's generate the ROC curve and learn its purpose at once.

The ROC curve is obtained through the `BinaryClassificationEvaluator` object. In listing 10.25, I create said

object, asking explicitly for the `areaUnderROC` metric, which yields the ROC curve.

Before getting started with the model performance metrics, let us create our evaluator. PySpark has one type of evaluator per model result type; since we are predicting a binary target, we use the `BinaryClassificationEvaluator`. Note that our evaluator takes the raw prediction as an input. The evaluator generates a single measure which is the area under the ROC curve, a number between zero and one (higher is better). We're doing great, but it would be nice to know what this number means…

Listing 10.25 Creating a `BinaryClassificationEvaluator` object and evaluating if

```
 language="python" linenumbering="unnumbered">from pyspark.ml.evaluation import Bina

trainingSummary = lrModel.summary

evaluator = BinaryClassificationEvaluator(
    labelCol="dessert",     (1)
    rawPredictionCol="rawPrediction",     (1)
    metricName="areaUnderROC",
)

accuracy = evaluator.evaluate(predictions)
print(f"Area under ROC = {accuracy} ")

# Area under ROC = 0.9927442079816466
```

(1) We pass our label (or target) and the rawPrediction column generated by our model. PySpark will usually be consistent with the nomenclature for the columns or objects it needs as a parameter (rawPredictionCol in this case).

A ROC curve is a plot of the true positive rate over the false positive rate. By changing the sensitivity of our model, we get two new true/false-positive rates, which we plot on a chart. A perfect model would have a 100% true positive rate and a 0% false-positive rate, which is the top left corner of our curve. To have a sense numerically of how close we are to this, the *area under the ROC curve* is used: it's the ratio of the chart that is under our ROC curve. In figure 10.3, the area under the curve is shaded: as we can see, an AUC (area under the curve) score of 0.9929… means there is not much above the curve.

Listing 10.26 Using matplotlib to display the ROC curve.

```
 language="python" linenumbering="unnumbered">import matplotlib.pyplot as plt

plt.figure(figsize=(5, 5))
plt.plot([0, 1], [0, 1], "r--")
plt.plot(
    lrModel.summary.roc.select("FPR").collect(),
    lrModel.summary.roc.select("TPR").collect(),
)
plt.xlabel("False positive rate")
plt.ylabel("True positive rate")
plt.show()
```

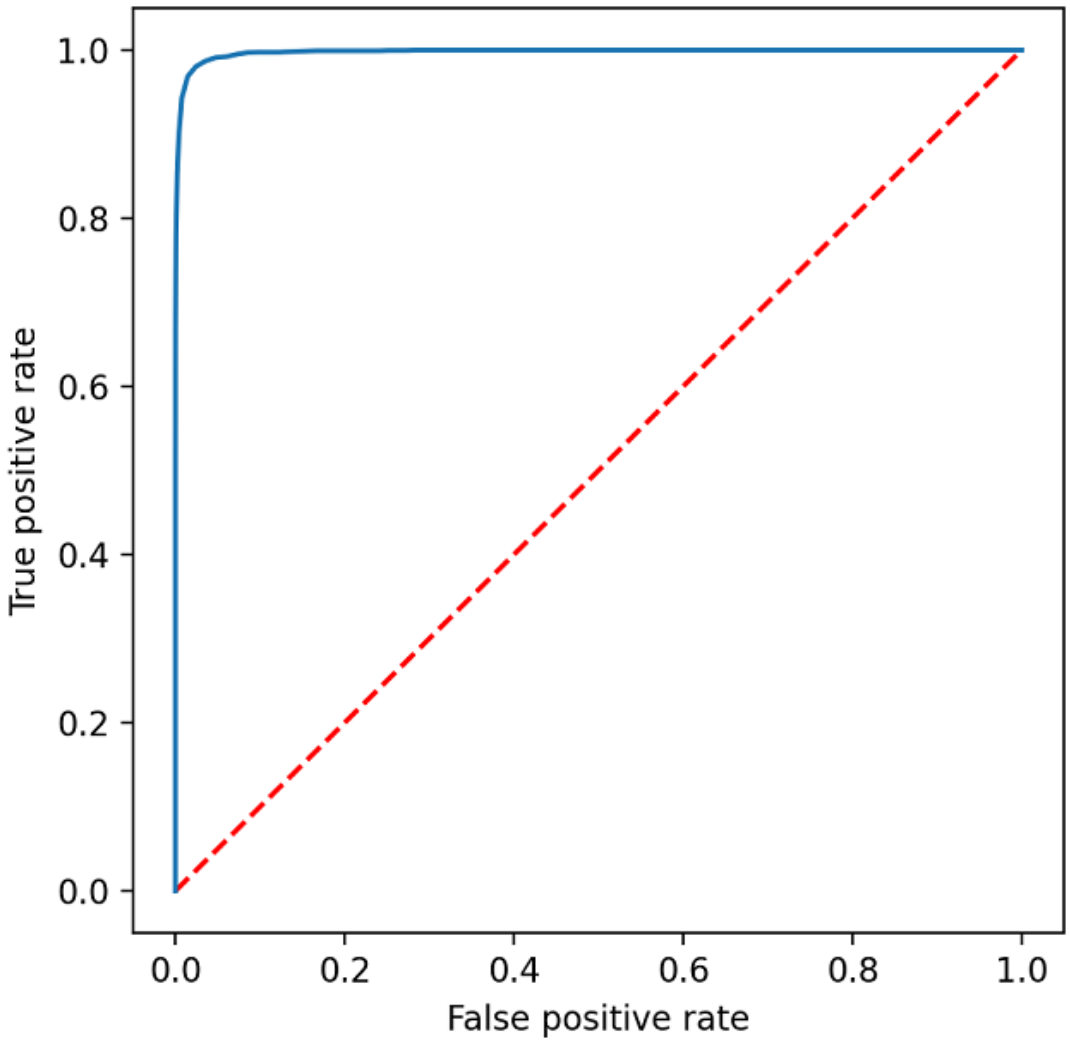


Figure 10.3 The ROC curve for our model. We want the blue curve to hit the top left corner as much as possible.

For the first run, our model did amazing. This does not mean our job is over:

initial model fitting is the first step in having a production-ready model. Our code looks a little artisanal and could use a cup or two of robustness. We also need to make sure our model will stay accurate as time goes by, which is why having an automated metrics pipeline is important.

The last section will slightly brush on model interpretability. We explore the coefficients of the model's features and discuss some improvements based on our findings.

10.3.2 Getting the biggest drivers from our model: extracting the coefficients

This section covers the extraction of our model features and their coefficients. We use those coefficients to get a sense of the most important features of the model and plan some improvements for a second iteration.

In 10.2.5, I explained that the features in a vector build via the `VectorAssembly` object keeps the feature names in the column's metadata dictionary and showed how to access them. We can access the schema through the `StructField` (see chapter 6 for a deep-dive into the schema and the `StructField`) contained in the top-level `StructField`. From them, it's just a little key-golfing. In listing 10.27, I show the metadata for the `features` column and extract the relevant fields. I then extract the coefficients from my `lrModel` through the `coefficient.values` attribute.

> **TIP** If you explicitly do binary feature engineering (such as one-hot encoding), PySpark will store the metadata for the features in a `binary` key instead of `numeric`. In our case, since we knew the features were binary, we did not treat them any special and PySpark lumped them in the numeric metadata key.

Listing 10.27 Extracting the feature names from the `features` vector and assembling them with their coefficients.

```
 language="python" linenumbering="unnumbered">feature_names = ["(Intercept)"] + [
 ❶
    x["name"]
    for x in (
        food_model
        .schema["features"]
        .metadata["ml_attr"]["attrs"]["numeric"]
    )
]
```

```
feature_coefficients = [lrModel.intercept] + list(     ❶
    lrModel.coefficients.values
)


coefficients = pd.DataFrame(
    feature_coefficients, index=feature_names, columns=["coef"]
)

coefficients["abs_coef"] = coefficients["coef"].abs()     ❷

print(coefficients.sort_values(["abs_coef"]))

#                               coef   abs_coef
# nutmeg                    0.002563   0.002563
# simmer                   -0.006231   0.006231
# fig                       0.007262   0.007262
# fat_ratio                 0.014916   0.014916
# pecan                     0.026738   0.026738
# ...
# chicken                 -14.488297  14.488297
# pork_chop               -14.696056  14.696056
# poppy                   -15.338339  15.338339
# horseradish             -16.199857  16.199857
# goose                   -17.784216  17.784216
```

❶ I add the intercept (the feature-less parameter of the model) manually since PySpark keeps it under its own intercept slot.
❷ Since negative and positive values are equally important in logistic regression, I create an absolute value column for easy ordering.

In 10.2.4, I explained that having features on the same scale (here from zero to one]) helps with the interpretability of the coefficients. Here, by ordering them by their absolute value, we can see which coefficients have the greatest impact on our model.

A coefficient close to zero, like `nutmeg`, `simmer`, and `fig`, means that this feature is not very predictive of our model. On the flip side, a very high or very low coefficient, like `goose`, `horseradish`, and `pork_chop`, means that this feature is highly predictive. When using a linear model, a positive coefficient means that the feature will predict towards a `1.0` (the dish is a dessert). Our results are not too surprising; looking at the very negative features, it seems that the presence of meat means "not a dessert"!

In the real world, our job would just start, but we are concluding this chapter on this bitter-sweet note. In this chapter, we have taken a data set, performed a summary cleaning on it, analyzed our data for correlation and dependency between

features, created custom features, assembled those features into a vector that we fed to a logistic regression model, and finally evaluated the results. This is a lot of steps, but you'll find that we leverage a lot of the skills acquired when learning about data manipulation.

Unsurprisingly, data science revolves very much around ones' ability to process, extract information, and apply the right abstractions on the data. Statistical know-how comes very usefully when you need to know the *why?* of specific models. PySpark provides functionality for a growing number of models, but each one will use a similar estimator/transformer setup. Now that you understand how the different components work, applying a different model will be a *piece of cake*!

10.4 Summary

- A big part in creating a machine learning model resides in data manipulation. For this, everything we've learned within `pyspark.sql` can be leveraged.
- The first step in creating an ML model is assessing data quality and addressing potential data problems. Moving from large data sets in PySpark to small summaries in pandas or plain Python speeds up data discovery and assessment.
- Feature creation and selection can either be done manually using the PySpark data manipulation API or by leveraging some `pyspark.ml` specific constructors, like the correlation matrix and the scaling transformer.
- Before training a model, every feature needs to be assembled in a vector using the `VectorAssembler` transformer.
- PySpark provides a unified interface for training and evaluating a model: you `fit()` on the training set and `transform()` on the testing set.
- PySpark provides useful metrics for model evaluation through a set of evaluator objects. You select the appropriate one based on your type of prediction (binary classification = `BinaryClassificationEvaluator`)

Appendix A: Exercise solutions

This appendix contains the solutions to the exercises present in the book.

CHAPTER 4

EXERCISE 4.1

Answer:

```
sample = spark.read.csv("sample.csv",
                        sep=",",
                        header=True",
                        quote="$",
                        inferSchema=True)
```

Explanation:

- `sample.csv` is the name of the file we want to ingest.
- The record delimiter is the comma. Since we are asked to provide a value there, I pass the comma character `,` explicitly, knowing it is the default one.
- The file has a header row, so I input `header=True`
- The quoting character is the dollar sign character, `$`, so I pass it as an argument to `quote`.
- Finally, since inferring the schema is nice, I pass `True` to `inferSchema`.

EXERCISE 4.2

Answer:

```
DIRECTORY = "../../data/Ch04"
logs_raw = spark.read.csv(os.path.join(DIRECTORY, "BroadcastLogs_2018_Q3_M8.CSV"),)

logs.printSchema()
# root
#  |-- _c0: string (nullable = true)

logs.show(5)
# +--------------------+
# |                 _c0|
# +--------------------+
# |BroadcastLogID|Lo...|
# |1196192316|3157|2...|
# |1196192317|3157|2...|
# |1196192318|3157|2...|
```

```
# |1196192319|3157|2...|
# +------------------+
# only showing top 5 rows
```

Two major differences:

1. PySpark globbed everything into a single string column, since it did not encounter the default delimiter (`,`) consistently in the records.
2. It named the record `_c0`, the default convention when it has no information about column names.

EXERCISE 4.3

Answer:

```
logs_clean = logs.select(*[x for x in logs.columns if not x.endswith("ID")])
```

Explanation:

I use the list comprehension trick on the data frame's columns, using the filtering clause `if not x.endswith("ID")` to keep only the columns that do not end with "ID".

EXERCISE 4.4

Answer:

c

Explanation:

Both `item` and `UPC` match a columns, while `prices` doesn't. PySpark will ignore the non-existent columns passed to `drop()`.

CHAPTER 5

EXERCISE 5.1

Answer:

`left`

Explanation:

A `left_semi` join only keep the records in left where the `my_column` value is also present in the `my_column` column in `right`. A `left_anti` join is the opposite: it keep the recorts not present. Unioning those two together results in the original data frame, `left`.

EXERCISE 5.2

Answer:

```
left.join(right, how="left",
          on=left["my_column"] == right["my_column"]).where(
          right["my_column"].isnull()
          ).select(left["my_column"]).
```

Explanation:

When performing an inner join, all the records from the left data frame are kept in the joined data frame. If the predicate in unsuccessful, then the column values from the right table are all set to null for the affected records. We just have to filter to keep only the unmatched records and then

select the `left["my_column"]` column.

EXERCISE 5.3

Answer:

```
left.alias("l").join(right.select("my_column").distinct().alias("r"), how="left",
          on=F.col("l.my_column") == F.col("r.my_column")).where(
          ~F.col("r.my_column").isnull()
          ).select(F.col("l.my_column"))
```

Explanation:

This follows the same pattern as the left anti join, but with a few more tricks.

A left semi join is equivalent to keep only the left records that resolve the predicate. Because a left join would duplicate left records if they are matched more than once in the right table, we have to remove the potentially duplicate values.

We can't `distinct()` at the end, since this would remove the duplicate values in the left table that we want to keep.

CHAPTER 8

EXERCISE 8.1

Answer:

I will create a sample data frame before providing the answer, to see it in action.

```
# I assume spark is initialized and that `spark` exists as a variable
from operator import add

exo_rdd = spark.sparkContext.parallelize(list(range(100)))

# Answer below
exo_rdd.map(lambda _: 1).reduce(add)
```

Explanation:

I start by mapping each element to the value 1, regardless of the input. The _ in the lambda function doesn't bind the elements because we don't process the element: we just care that they exits. After the map operation, we have a RDD containing only the value 1. We can `reduce(sum)` to get the sum of all the ones, which yields the number of elements in the RDD.

EXERCISE 8.2

Answer:

a

Explanation:

Filter will drop any values when the predicate (the function passed as an argument) returns a falsey value. In Python, 0, `None`, and empty collections are falsey. Since the predicate returns the value unchanged, `0`, `None`, `[]` and `0.0` are falsey and filtered out, leaving only `[1]` as the answer.

Appendix B: Installing PySpark locally

SIDEBAR **Last update: 2020-06-21**

Python 2 is now officially unsupported as of January 1st, 2020. At the moment, most OSes are in the transitional period between Python 2 and 3, which is why I spend a little time discussing how to install Python 3. I expect this guide to become simpler as time goes.

I am currently targeting the most recent version of each OS as of time of last update.

- Windows 10
- OS.X Catalina
- GNU/Linux Ubuntu 20.04 LTS

This appendix covers the installation of standalone Spark and PySpark on your own computer, whether it's running Windows, Os.X or Linux.

Having a local PySpark cluster means that you'll be able to experiment with the syntax, using smaller data sets. You don't have to acquire multiple computers or spend any money on managed PySpark on the cloud until you're ready to scale your programs.

Spark is a complex piece of software and most guides out there are over-complicating the installation proces. We'll take a much simpler approach by installing the bare minimum to start, and building from there. Our goals are as follow:

- Install Java (Spark is written in Scala, which runs on the Java Virtual Machine, or JVM).
- Install Spark
- Install Python 3 and IPython
- Launch a PySpark shell using IPython
- (Optional): Install Jupyter and use it with PySpark.

B.1 Windows

When working on Windows, you either have the option to install Spark directly on Windows, or to use WSL (Windows Subsystem for Linux). If you want to use WSL, follow the instructions at aka.ms/wslinstall and then follow the instructions for GNU/Linux. If you want to install on plain Windows, follow the rest of this section.

B.1.1 Install Java

The easiest way to install Java on Windows is to go on www.java.com and follow the download and installation instructions for downloading Java 11. Make sure to read the installer steps to avoid installing non-useful software!

> **WARNING** If you are installing Spark 2.4.6 or before, you will need to install Java 8 instead of Java 11.

B.1.2 Install 7-zip

Spark is available as a GZIP archive (`.tgz`) file on their website. By default, windows doesn't provide a native way to extract those files. The most popular option is 7-zip[8]. Simply go on the website, download the program and follow the installation instructions.

B.1.3 Download and install Apache Spark

Go on the Apache website and download the latest Spark release. You shouldn't have to change the default options, but Figure-B.1 displays the ones I see when I navigate to the download page. Make sure to download the signatures and checksums if you want to validate the download (step 4 on the page).

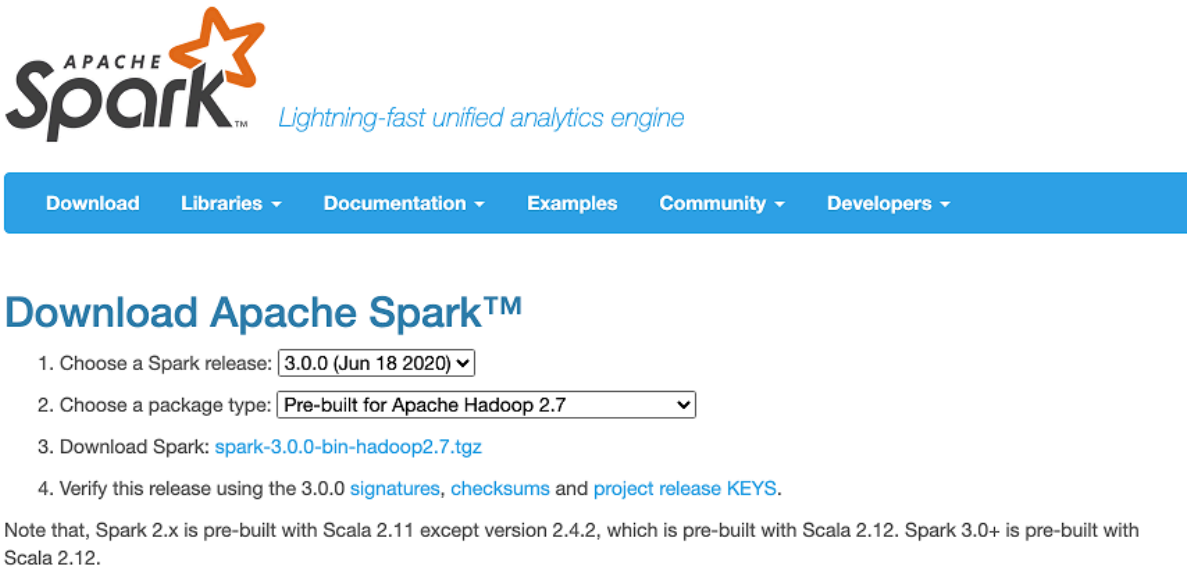


Figure B.1 The options to download Spark

Once you have downloaded the file, unzip the file using 7-zip. I recommend putting the directory under `C:\Users\[YOUR_USER_NAME]\spark` .

Next, we need to download a `winutils.exe` to prevent some cryptic Hadoop errors. Go on the github.com/cdarlint/winutils repository and download the `winutils.exe` file in the `hadoop-2.7.X/bin` directory where X is the highest number. As of the time of writing, it is 2.7.7. Keep the `README.md` of the repository handy.

> **TIP** Spark is also available with a more recent Hadoop, but it is not the default option. If you select "Pre-built for Apache Hadoop 3.2 and later" in step 2 of the download form, take the most recent version of winutils.

Place the `winutils.exe` in the `bin` directory of your Spark inrtallation. Then, set the environment variables as listed on the winutils repository's `README.md` To do so, open the start menu and search for "Edit the system environment variables". Click on the "Environment variables button" (see Figure-B.2) and then add them there. You will also need to set `SPARK_HOME` to the directory of your Spark installation (`C:\Users\[YOUR-USER-NAME]\spark\bin`). Finally, add the `%SPARK_HOME%\bin` directory to your `PATH` environment variable.

> **NOTE** For the `PATH` variable, you might already have some in there. If this is the case, double click on the variable and append `%HADOOP_HOME%\bin` to the list, as well as `%SPARK_HOME%\bin`.

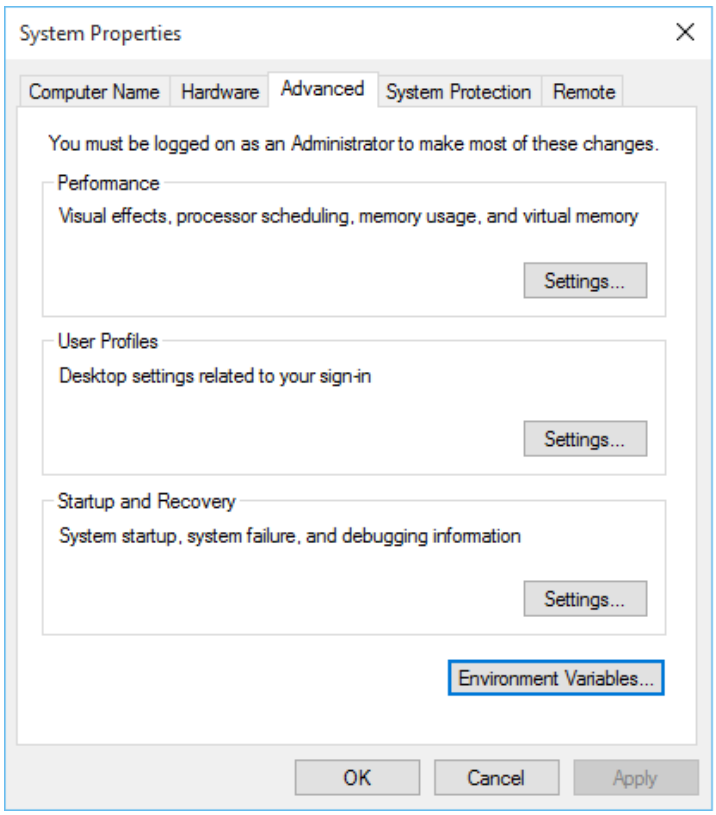


Figure B.2 Setting environment variables for Hadoop on Windows

B.1.4 Install Python

The easiest way to get Python 3 is to use the Anaconda Distribution. Go to www.anaconda.com/distribution and follow the installation instructions, making sure you're getting the 64-bits Graphical installer for Python 3.X for your OS.

Once Anaconda is installed, we can activate the Python 3 environment by selecting the "Anaconda Powershell Prompt" in the start menu. If you want to create a dedicated virtual environment for PySpark, use the following command.

```
$ conda create -n pyspark python=3.8 pandas pyspark=3.0.0
```

WARNING Python 3.8 is supported only using Spark 3.0. If you use Spark 2.4.X or before, be sure to specify Python 3.7 in you environment creation.

Then, to select your newly created environment, just input `conda activate pyspark` in the Anaconda Prompt.

B.1.5 Launching an iPython REPL and starting PySpark

If you have configured the `SPARK_HOME` and `PATH` variables, your Python REPL will have access to a local instance of pyspark. Follow the next code block to launch iPython.

> **TIP** If you aren't comfortable with the Command Line and Powershell, I've personally learned to use it using *Learn Powershell in a Month of Lunches* by Don Jones and Jeffery D. Hicks (Manning, 2016).

```
conda activate pyspark ❶
ipython
```

❶ Only if you have created a `pypark` virtual environment.

Then, within the REPL, you can import pyspark and get rolling.

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.getOrCreate()
```

> **NOTE** Spark provides a `pyspark.cmd` helper command through the `bin` directory of your Spark installation. I prefer accessing PySpark through a regular Python REPL when working locally as I find it easier to install libraries and know exactly which Python you're using. It also interfaces well with your favorite editor.

B.1.6 (Optional) Install and run Jupyter to use Jupyter notebook

Since we have configured PySpark to be discovered from a regular Python process, we don't have any further configuration to do to use it with a notebook. In your Anaconda Powershell window, install jupyter using the following command.

```
conda install -c conda-forge notebook
```

You can now run a Jupyter notebook server using the following command.

```
jupyter notebook
```

B.2 macOS

With macOS, the easiest option — by far — is to use the HomeBrew `apache-spark` package. It takes care of all dependencies (I still recommend using Anaconda for managing Python environments, for simplicity).

B.2.1 Install Homebrew

HomeBrew is a package manager for OS.X. It provides a simple command line interface to install many popular software packages and keep them up to date. While you can follow the manual "download and install" steps you'll find on the Windows OS with little change, Homebrew will simplify our installation process to a few commands.

To install Homebrew, go to brew.sh and follow the installation instructions. You'll be able to interact with Homebrew through the `brew` command.

B.2.2 Install Java and Spark

Input the following command in a terminal.

```
$ brew install apache-spark
```

You can specify the version you want; I recommend getting the latest by passing no parameters.

B.2.3 Install Anaconda/Python

The easiest way to get Python 3 is to use the Anaconda Distribution. Go to www.anaconda.com/distribution and follow the installation instructions, making sure you're getting the 64-bits Graphical installer for Python 3.X for your OS.

```
$ conda create -n pyspark python=3.8 pandas pyspark=3.0.0
```

If it's your first time using Anaconda, follow the instructions to register your shell.

> **WARNING** Python 3.8 is supported only using Spark 3.0. If you use Spark 2.4.X or before, be sure to specify Python 3.7 in you environment creation.

Then, to select your newly created environment, just input `conda activate pyspark` in the Anaconda Prompt.

B.2.4 Launching a iPython REPL and starting PySpark

Homebrew should have the `SPARK_HOME` and `PATH` environment variables, so your Python REPL will have access to a local instance of pyspark. You just have to type the following.

```
conda activate pyspark ❶
ipython
```

❶ Only if you have created a `pypark` virtual environment.

Then, within the REPL, you can import pyspark and get rolling.

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.getOrCreate()
```

B.2.5 (Optional) Install and run Jupyter to use Jupyter notebook

Since we have configured PySpark to be discovered from a regular Python process, we don't have any further configuration to do to use it with a notebook. In your Anaconda Powershell window, install jupyter using the following command.

```
conda install -c conda-forge notebook
```

You can now run a Jupyter notebook server using the following command.

```
jupyter notebook
```

B.3 GNU/Linux and WSL

B.3.1 Install Java

Most GNU/Linux distributions provide a package manager. OpenJDK version 11 is available through the software repository.

```
`sudo apt-get install openjdk-11-jre`
```

WARNING If you want to install a version of Spark prior to 3.0.0, install `openjdk-8-jre` instead.

B.3.2 Installing Spark

Go on the Apache website and download the latest Spark release. You shouldn't have to change the default options, but Figure-B.1 displays the ones I see when I navigate to the download page. Make sure to download the signatures and checksums if you want to validate the download (step 4 on the page).

TIP On WSL (and sometimes Linux), you don't have a graphical user interface really available. The easiest way to download Spark is to go on the website, follow the line, copy the link of the nearest mirror and past it along with `wget` command.

```
wget [YOUR_PASTED_DOWNLOAD_URL]
```

If you want to know more about using the command line on Linux (and Os.X) proficiently, a good free reference is *The Linux Command Line* by William Shotts[9]. It is also available as a paper or e-book (No Starch Press, 2019).

Once you have downloaded the file, unzip the file (using 7-zip on Windows). If you are using the

command line, the following command will do the trick. Make sure you're replacing the `spark-[…].gz` by the name of the file you just downloaded.

```
tar -xvzf spark-[...].gz
```

This will unzip the content of the archive into a directory. You can now rename and move the directory to your liking. I usually put it under `/home/[MY-USER-NAME]/bin/spark-3.0.0/` (and rename if the name is not identical) and the instructions will use that directory.

Set the following environment variables.

```
echo 'export SPARK_HOME="$HOME/bin/spark-3.0.0"' >> ~/.bashrc
```

B.3.3 Install Python 3 and IPython

Python 3 is already provided, you just have to install IPython. Input the following command in a terminal.

```
sudo apt-get install ipython3
```

> **TIP** You can also use Anaconda on GNU/Linux! Follow the instructions on the macOS section.

B.3.4 Launch PySpark with IPython

Launch an iPython shell.

```
ipython
```

Then, within the REPL, you can import pyspark and get rolling.

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.getOrCreate()
```

B.3.5 (Optional) Install and run Jupyter to use Jupyter notebook

Since we have configured PySpark to be discovered from a regular Python process, we don't have any further configuration to do to use it with a notebook. In your terminal, input the following to install jupyter.

```
pip install notebook
```

You can now run a Jupyter notebook server using the following command.

```
jupyter notebook
```

Notes

1. It can be a fun probability exercise to compute by how much, but I will try to keep the math stuff at a minimum.

2. Application Programming Interface, which is basically the set of functions, classes and variables provided for you to interact with

3. Java Virtual Machine, which is like an emulator running on your computer. Both Java and Scala targets the JVM.

4. It does actually a whole lot more, but we will cover other aspects in Chapter 7.

5. writing a program using the lowest possible number of characters.

6. In Quebecois, we say "s'enfarger dans les fleurs du tapis" to talk of someone who's too bogged down on the details. Transliterated, it would be "to trip over the rug's flowers".

7. On windows, you might sometimes have issues with the pip wheels. If this is your case, refer to the PyArrow documentation page for installing: arrow.apache.org/docs/python/install.html

8. www.7-zip.org/

9. linuxcommand.org/