

Python data analysis at scale

PySpark

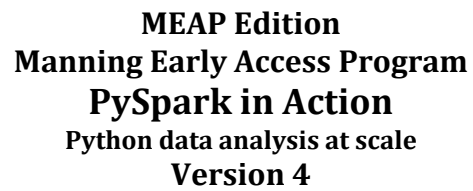
IN ACTION

Jonathan Rioux

MEAP



MANNING



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

brief contents

PART 1: WALK

- 1 Introduction*
- 2 Your first data program with PySpark*
- 3 Submitting and scaling your first PySpark program*
- 4 Analyzing data using pyspark.sql*

PART 2: JOG

- 5 It happens to the best of us: cleaning messy data*
- 6 Making sense of your data: types, structure, and semantics*
- 7 Bilingual PySpark: blending Python and SQL code*
- 8 Faster big data processing: a primer*
- 9 Extending PySpark's capacities with user defined functions*

PART 3: RUN

- 10 A foray into machine learning: logistic regression with PySpark*
- 11 Simplifying your experiments with Machine learning pipelines*
- 12 Machine learning for unstructured data*
- 13 PySpark for graphes: GraphFrames*
- 14 Testing PySpark code*
- 15 Going full circle: structuring end-to-end PySpark code*

APPENDIXES

- A Installing PySpark on your computer*

B Using PySpark with a cloud provider

C Python essentials

D Using PySpark with a notebook UI

E Efficiently using PySpark's API documentation

welcome

Thank you for purchasing the MEAP for *PySpark in Action: Python data analysis at scale*. 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

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 Go, 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.

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 API² 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 be 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 reads 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 A covers how to get your own local Spark running, while Appendix B 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 JVM³ 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.

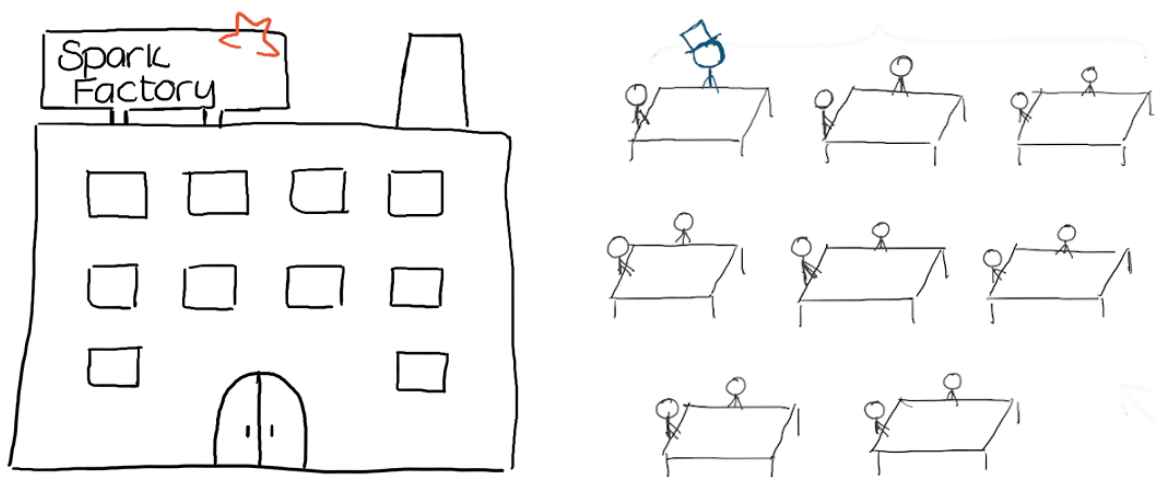


Figure 1.1 A totally relatable data factory, outside and in.

If we look at 1.1, 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 1.1 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 1.1, 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 C. 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 A) or using a cloud provider (Appendix B). 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.

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 A contains step-by-step instructions for Linux, OSX and Windows. If you can't install it on your computer, or prefer not to, Appendix B 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. 2.1 shows what happens on my local machine.

- 1 When using PySpark locally, you most often won't have a full Hadoop cluster pre-configured. For learning purposes, this is perfectly fine.
- 2 Spark is indicating the level of details it'll provide to you. We will see how to configure this in 2.1.
- 3 We are using Spark version 2.4.3
- 4 PySpark is using the Python available on your path.
- 5 The `pyspark` shell provides an entry point for you through the variable `spark`. More on this in 2.1.1.
- 6 The REPL is now ready for your input!

While all the information provided in 2.1 is useful, two elements are worth expanding on: the `SparkSession` entry point and the log level.

In 2.1 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 PySpark⁴.

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 2.2.

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

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

2.1 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 <code>FATAL</code> 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 2.1

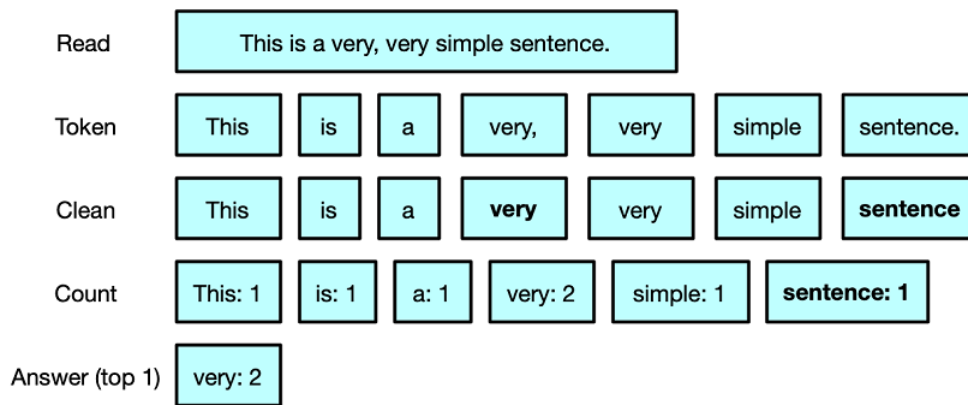


Figure 2.1 A simplified flow of our program, illustrating the 5 steps.

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.

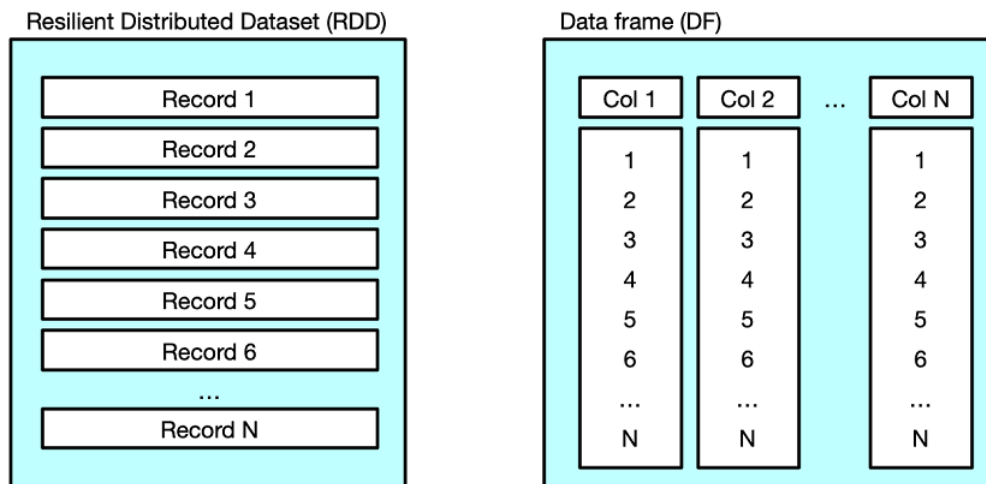
2.3 Reading and ingesting data into a data frame

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 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. 2.2 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.



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 2.3 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 0x115belb00>

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 as 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 2.3. 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.

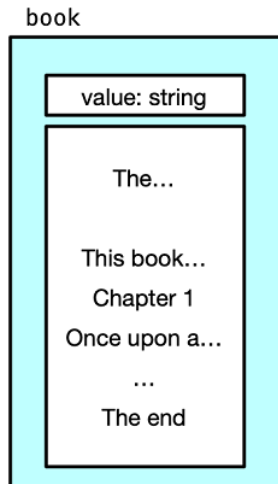


Figure 2.3 A high-level schema of a our book data frame, containing a value string column. We can see the name of the column, its type, and a small snippet of the data.

If you want to see the schema in a more readable way, you can use the handy method `printSchema()`, illustrated in ["ch02-printschemalisting-id"](#). 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 2.3.

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 2.3, we saw that the default behaviour of inputting 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 2.7 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 2.8 shows a couple options, starting 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 2.9.

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()`.


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

PySpark provides more than one way to select columns. I displayed the three most common in 2.11.

```
from pyspark.sql.functions import col

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

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.

WARNING There is theoretically a fourth way to select a column, which is by passing the column name as a simple string. In this case, you'd just have to write `book.select('value')`.

For simple select statements (and other methods that I'll cover later), it can be a viable option. That being said, it's not as flexible as the other options and the moment that your code requires column transformations, like in 2.4, you'll have to use another option. Future-proof your code by picking up one of the three from the start.

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. 2.12 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([('abl2cd',)], ['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 2.13.

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. 2.14 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 2.4 explains the process.

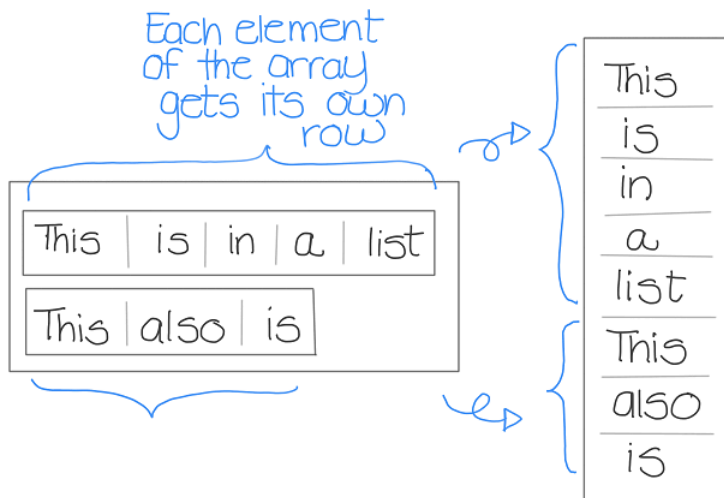


Figure 2.4 Exploding a data frame of `array[String]` into a data frame of `String`. Each element of each array becomes its own record.

The code follows the same structure as `split`, and you can see the results in 2.15. 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.

2.16 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 2.5.

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


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 (regexpr.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 2.18, 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

[Practical] 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")
)
```

[Conceptual] Exercise 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"
)
```

[Practical] Exercise 2.3

Let's take our `words_nonull` data frame, available in 2.18. 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.

[Practical, Beyond] Exercise 2.4

The `where` clause takes a Boolean expression over one or many column to filter the data frame (see 2.7). 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_nonnull` data frame (see exercise 2.3). Write the code to do so.

[Conceptual] 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"))
```

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 listing 3.1 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 figure 3.1

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
```

words_nonull: DataFrame

word
online
some
online
some
some
still
...
cautious

groups = words_nonull.groupby("word"): GroupedData




word	
online	
some	
...	...
cautious	

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

Peeking at the `results` data frame in listing 3.1, 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.

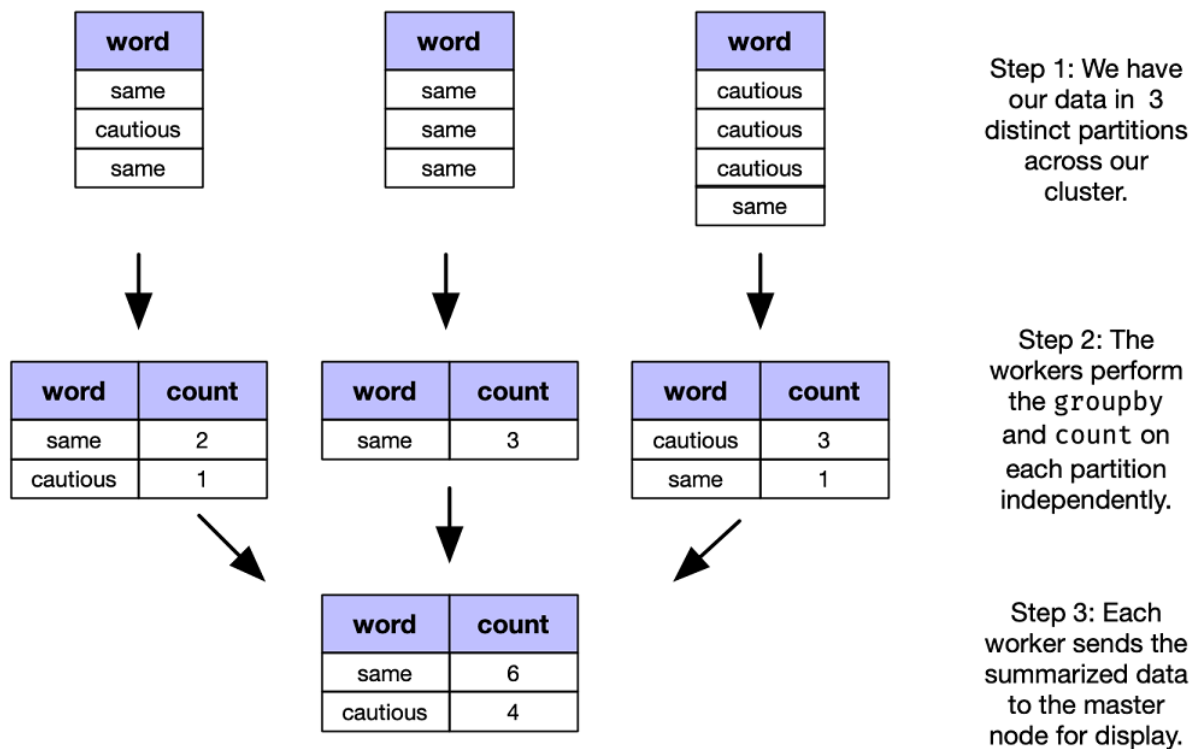


Figure 3.2 A distributed group by on our `words_nonull` data frame. The work is performed in a distributed fashion until we need to assemble the results in a cohesive display, via `show()`.

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 3.1, 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)

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

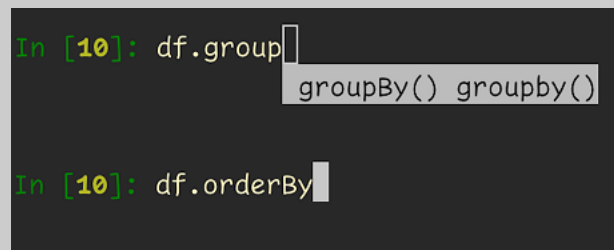
```
# +---+ +---+ +
# |word|count|
# +---+ +---+ +
# | the| 4480|
# |  to| 4218|
# |  of| 3711|
# | and| 3504|
# | her| 2199|
# |  a | 1982|
# |  in| 1909|
# | was| 1838|
# |  i | 1749|
# | she| 1668|
# +---+ +---+ +
# only showing top 10 rows
```

©Manning Publications Co. To comment go to liveBook
Licensed to Mike Rumore #3100, vppall@gmail.com

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.



```
In [10]: df.groupby
          groupBy() groupby()

In [10]: df.orderBy
```

Figure 3.3 PySpark's camelcase vs camelCase

Part of this incoherence is due to Spark's heritage. Scala prefers camelCase for methods. On the other hand, we saw `regex_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.

Listing 3.3 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
```

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. Listing 3.4 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 listing 3.5. 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_nonnull = words_clean.where(col("word") != "")

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

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

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

This program runs perfectly if 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. Listing 3.6 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 star 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.

SIDEBAR Make your life easier by using Python's parentheses

If you look at the "after" code of listing 3.7, 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 (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 golfer⁵ 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 listing 3.9, 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 listing 3.10, 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 occurrences from a book.")
        .getOrCreate())
```

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

We saw at the beginning of 3.5 how to submit a program using `spark-submit`. Let's try again, in listing 3.11, 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 B 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. 3.12 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 3.13.

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 `coalesce(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`)

[Practical] Exercise 3.1

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

[Practical] 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*.

[Practical, Beyond] Exercise 3.3

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

[Conceptual] 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 data with pyspark.sql

This chapter covers

- Reading delimited data into a PySpark data frame
- Ingesting and exploring tabular or relational data
- Selecting, manipulating, renaming and deleting columns in a data frame
- Performing simple joins between two data frames
- Grouping data and computing summary on a data frame

So far, in Chapter 2 and 3, we've dealt with textual data, which is *unstructured*. Through a chain of transformations, we extracted some information in order 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. We'll build on our prior knowledge, push the envelope a little further, and by the end, you'll know how to wrangle the most common type of data to answer your own questions.

4.1 What is tabular data?

Tabular data is data that we can represent in a 2-dimensional table. You have rows and columns containing a single (or *simple*) value. A good example would be a 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. Figure 4.1 provides an example of a small grocery list. We have the three columns mentioned, as well as 4 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 A grocery list as tabular data, with 4 records/rows and 3 columns

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. PySpark has a slight different way to map tabular data into its model. In Chapter 2, we saw that PySpark operates on data via transformations on either the whole data frame's structure (for instance, using `groupby` to translate it into `GroupedData` object) or on some of the columns objects (like when you're using a function like `split`). This is still the case for tabular data and in fact will be the case for all the data representations PySpark abstracts into its data frame structure.

4.1.1 How does PySpark represents 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 a other words, we took *unstructured* data (a body of text), performed some transformations and created a two column table containing the information we wanted.

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. We could have done the same with `pandas`, but this would have required to map the data to a

pandas data frame first. Considering the size of our grocery list, avoiding an additional import as well as a little boilerplate is something I'm glad to avoid.

Listing 4.1 Creating a data frame out of our grocery list

```
language="python" linenumbering="unnumbered">my_grocery_list = [
    ["Banana", 2, 1.74],
    ["Apple", 4, 2.04],
    ["Carrot", 1, 1.09],
    ["Cake", 1, 10.99],
]

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

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

We can easily create a data frame from data in our program with the `spark.createDataFrame` function as listing 4.1 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'll cover in Chapter 6. The second parameter is the *schema* of the data frame. Chapter 5 will cover automatic and manual schema definition 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 figure 4.2. Each column maps to data stored somewhere on our cluster, managed by PySpark. We operate on the abstract structure and let the optimizer do its job.

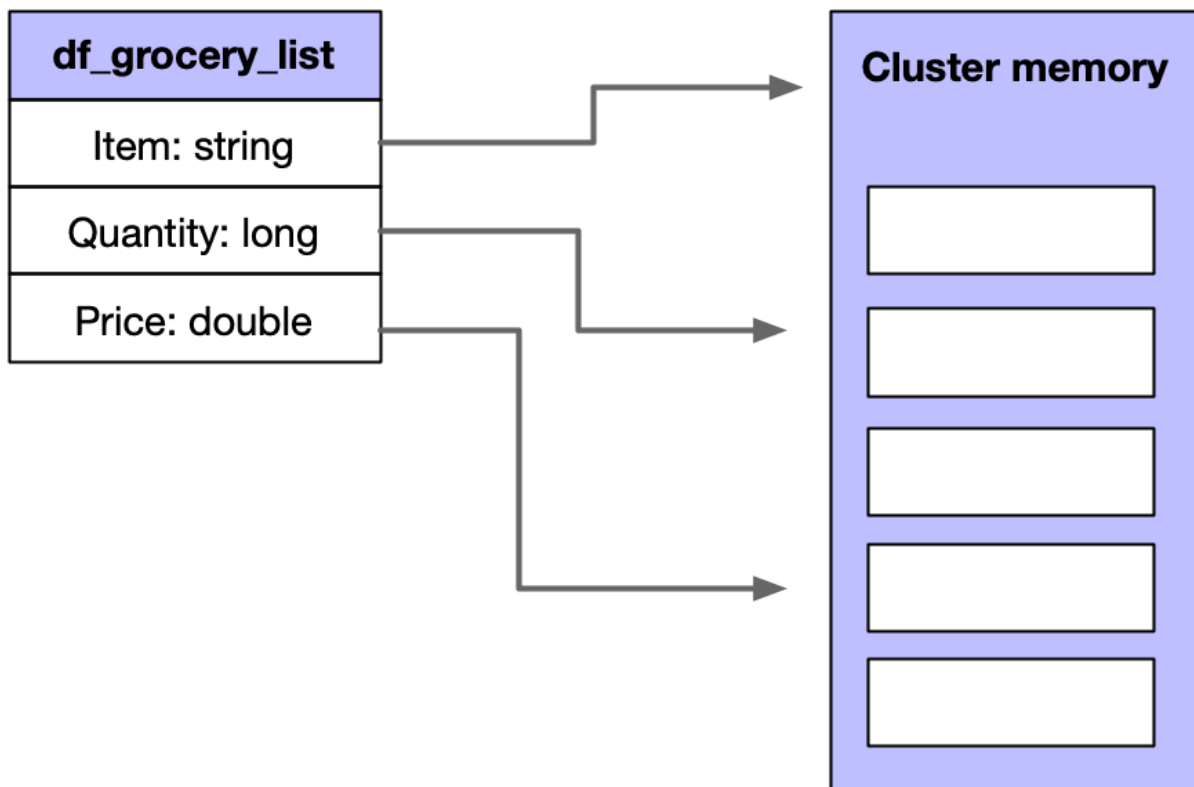


Figure 4.2 A graphical representation of our `df_grocery_list` data frame

There is nothing special about our data frame. PySpark gladly represented our tabular data using our column definitions. This means that all the functions we learned so far are applicable to our tabular data. By having one flexible structure for many data representation —we’ve done text and tabular so far —PySpark makes it easy to move from one domain to another. You don’t have to ask yourself which structure is best: use the data frame!

4.2 PySpark for analyzing and processing 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 our eyes can process 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 a 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. 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 capacities, and doesn't play with other charting libraries (like matplotlib, seaborn or altair for instance). When you think about it, 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'll provide the code I used to generate them, but instead of explaining the process each time, I'll provide explanations —as well as a primer in using pandas with PySpark—in Appendix C.

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 portal¹, 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 will contain 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. Once again, the examples are assuming that the data is downloaded under `data/Ch03` and that you've launched PySpark from the root of the book's repository.

Footnote 1 <https://open.canada.ca/data/en/dataset/800106c1-0b08-401e-8be2-ac45d62e662e>

TODO: Change the setup of the directory in the code file TODO: Put a sample of the data under the book's repository

4.3 Reading delimited data in PySpark

Before we can start analyzing our data, we have to ingest it. In Chapter 2, our textual data was pretty evident and we didn't have to use any options. With a delimiter separated value (or DSV) file, the work is a little more complicated. We have to ask ourselves the following questions.

1. What is the delimiter? Is it a comma (CSV), a tab character (TSV) or another character?
2. Are the strings quoted (in case the delimiter is inside some of them)? Is it a single quote, a double quote?
3. Is the first row containing the name of the columns, or do we have to provide them separately?
4. What is the type of each column? DSV files don't provide any guidance if 9 is the number 9 or the character 9.

There are many other questions we should ask ourselves when reading a DSV file, and this is why PySpark provides a whopping 26 optional parameters with the `SparkReader.csv()` method. Those are definitely the 4 most common, and Chapter 4 will look at more of them when we talk about less-than-pristine data. Have a look at listing 4.2 and see if you can answer all four questions by looking at which options I used.

Listing 4.2 Reading our broadcasting information

```
language="python" linenumbering="unnumbered">import os

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

Let's now answer our questions one by one.

1. The delimiter being used by this data set is the vertical bar, or "|".
2. We aren't specifying an explicit for string quotation here. Our reader will use the default option, which is the double quote (").
3. The `header=True` means that the first row contains the column names.

4. We're relying on PySpark's schema inference capacities to determine the types of the columns.

Before exploring the data, let's walk through what the code does.

4.3.1 Customizing the *SparkReader* object to read DSV data

We saw in listing 4.2 that we can specialize the `SparkReader` to read DSV files by using the `csv` method. The choice of name is a little unfortunate, since CSV is a subset of what this method can do, but nowadays, most people use CSV to refer to any kind of delimited data. I'll trade some precision for the popular opinion.

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.

The first, and most important, option is the delimiter, which is defined by the parameter `sep` (for separator). By default, it's set to be the comma character, but you can specify any single character. This file uses the vertical bar ("`|`"), a usual favourite when dealing with data that can contain commas.

I am not explicitly passing a value for the `quote` parameter. We're relying on the default parameter, which is the double quote character ("`\"`"). In our document, string are double quoted to avoid confusing the parser if the delimiter is part of the string. 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.

The `header` 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 have the option to pass the column names explicitly (using the `schema` parameter). 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`, ...).

TIP

When it comes to multiple files, if you set `header=True`, PySpark will look at all the files for the header, so it'll work as long as one of them has it. I tried to trick it into getting confused, but it has been pretty consistent at doing what I expected.

Finally, we can rely on PySpark schema discovering capacity by passing `inferSchema=True`. This operation forces PySpark to go over all the files

twice: one time to set the types of each columns, one time to ingest the data. This makes the ingestion quite a bit longer, but avoids us to write the schema. 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 data. 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 PySpark-scale, you'll often won't get the chance to inspect each row one by one to fix mistakes. Chapter 5 will cover some strategies to ease the pain, and XREF big-data-file-format will show you some ways to share your data with the schema included.

If we look at our data schema, displayed on listing 4.3, you can see that most of the columns seem to have a reasonable type attached. That's plenty enough for us to get started with come exploration.

Listing 4.3 The schema of our `logs` data frame

```
language="python" linenumbering="unnumbered">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)
```

4.3.2 Exploring the shape of our data

It's very rare that you'll be working with a single table. More frequently, you'll have your data spread across multiple tables, and you'll need to merge them into a cohesive set. This is what I call the *shape* of the data: how are the multiple tables organized together? How do the records relate to one another? This section will introduce one specific, but common, type of shapes and will merge the data we need into a single table.

If we look at the structure of the data and at a few records (`logs.show()` will do the trick), we can see that there is a lot of ID numerical fields that don't mean a whole much. Looking at the data documentation, we can see that the table is shaped in what's called a *star schema*. We have our logs table at the centre, and every ID field maps to an ancillary table defining what each ID means. This common practice in the relational database world is called *normalization* and is used to avoid duplicating pieces of data and improve data integrity. In figure 4.3, the centre table contains IDs that map to the auxiliary tables around called *link*. Note that the quantity isn't normalized, since we're already storing it as a long integer. In other words, the *relations* are stored in the centre table and the definitions are in the *link* tables.

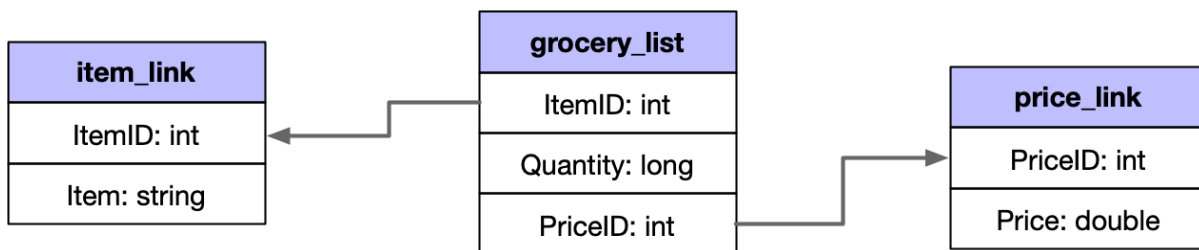


Figure 4.3 Our grocery list example as a very simple (2 relationships) star schema

In Spark's universe, we often prefer working with a single table instead of linking a multitude of tables to get the data. This is called *denormalized* tables, or colloquially *fat* tables. Before exploring how we can plump our table and make them more analytics-friendly, we'll work on assessing and cleaning the centre table first. Denormalization, on our case, would look like figure 4.4. We match the ID

field in both tables to create one that contains the information of both. Our grocery list becomes whole again. I illustrated just one example for each table to avoid cluttering the visual too much.

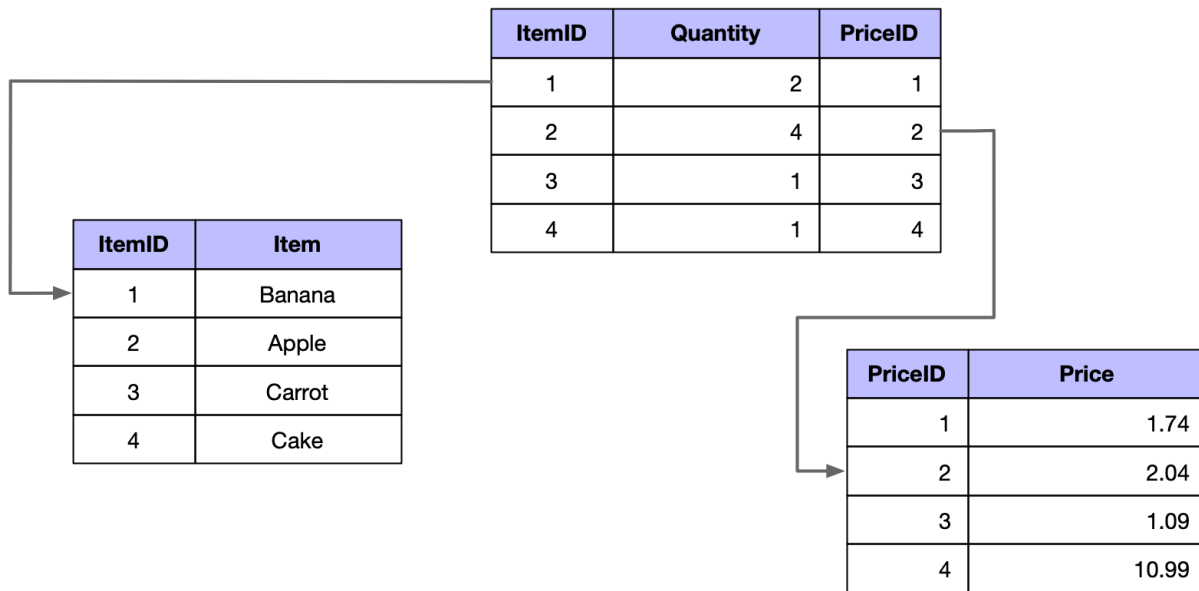


Figure 4.4 Denormalization of part of our grocery list.

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, it reduces the probability of getting anomalies or illogical records in your data.

When dealing with analytics, having many links to make in order to build our table might be cumbersome. 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 he'll provide you with the information to denormalize the tables, and Chapter 6 will show you how you can adapt the code into PySpark.

4.4 Manipulating structured data

It is common practice to explore and summarize the data when you first get acquainted with it. It's just like a first date: you want to understand what's inside without spending all your time over-focusing on details. This section will provide some very useful ways to slice and dice data, delete some element you don't need and rename the structure so it's friendlier to work with. No flowers required.

4.4.1 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 data². 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 output is garbled, because we are showing too many columns at once. Time to select our way to sanity.

Footnote 2 unless you're using eagerly evaluated Spark, but let's assume you're not.

As we saw in Chapter 2, at its simplest, `select()` can take a list of strings representing columns and will return a data frame containing only those columns. This way, we can keep our exploration tidy and check a few columns at the time. An example is displayed in listing 4.4.

Listing 4.4 Selecting 5 rows of the first 3 columns of our data frame

```
language="python" linenumbering="unnumbered">logs.select(logs.columns[:3]).show(5,
```

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
```

We already know 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.

Our `logs` data frame 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.

If you take a peek at PySpark's documentation, you can see that `select` only takes a single parameter, named `*cols`. This star is used in Python for unpacking collections. From a PySpark perspective, those lines are equivalent. 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 C will provide you with a good overview of its usage.

```
import pyspark.sql.functions as F

# 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, no need to iterate over your list. The API just became a little bit friendlier!

In the spirit of being clever (or lazy), let's expand our selection code to see every columns 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 listing 4.5 shows one of the way we can do it.

Listing 4.5 Peeking at the data frame in chunks of 3 columns

```
language="python" linenumbering="unnumbered">import numpy as np

column_split = np.array_split(logs.columns, len(logs.columns) / 3)
[logs.select(*x).show(5, False) for x in column_split]
```

Let's take each line one at a time.

We start by splitting the `logs.columns` list into approximate groups of 3. Numpy provides a ready-to-go function for this: we pass the list to `split` and the number of sections we wish. Since we're aiming for 3 columns at a time, we pass

`len(log.columns) / 3` as our wanted number of sections. This is a great example of how you can use Python code in your PySpark program. This is an incredibly interesting subject that Chapter 8 will cover in greater details.

The third line uses a list comprehension to show each group formed in `column_split`. List comprehensions are one of my favourite aspects of Python. If you aren't familiar with them, Appendix C will give you a great primer on this. In a nutshell, we iterate over `column_split`, unpacking each sub-group into the `select()` method, which we `show()` after.

This example shows how easy it is to blend Python code with PySpark. While the data frame API provides a ton of functions, it exposes information, such as column names, into convenient Python structures. I won't avoid using functionality from third-party libraries when it make sense, but like in listing 4.5, I'll do my best to explain what it does and why we're using it.

`show()` isn't the only game in town to display data on the screen. PySpark provides one additional method to summarize data in a easy to digest way. Enter `describe()`

4.4.2 Summarizing your data frame

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()` methods does exactly that. By default, it will show those statistics on all numerical and string columns, which we know will overflow on our screen and become unreadable. XREF ch03-describe show a saner way to do it, leveraging our recent play with list comprehensions over `logs.columns`. Note that `describe()` will (lazily) compute the data frame but won't display it, so we have to `show()` the result.

Listing 4.6 Describing everything in one fell swoop

```
language="python" linenumbering="unnumbered">[logs.describe(i).show() for i in logs
```

summary	BroadcastLogID
count	7169318
mean	1.2168893983502293E9
stddev	1.4982198462893466E7
min	1195788151

```
# |      max |      1249453843 |
# +-----+-----+
#
# [... many more little tables]
```

It will take more time than doing everything in one fell swoop, but the output will be a lot friendlier. We're using a list comprehension over the columns of our data frame, selecting and describing each one at a time. 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, 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()`), `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. Listing 4.7 shows how you can replace `describe()` for `summary()` into our list comprehension, and the result of doing so.

Listing 4.7 Summarizing everything in one fell swoop

```
language="python" linenumbering="unnumbered">[logs.select(i).summary().show() for i
# +-----+-----+
# |summary|      BroadcastLogID|
# +-----+-----+
# |  count |      7169318|
# |   mean |1.2168893983502293E9|
# | stddev |1.4982198462893466E7|
# |   min |      1195788151|
# |   25% |      1204698764|
# |   50% |      1213285847|
# |   75% |      1226221066|
# |   max |      1249453843|
# +-----+-----+
#
# [... many more slightly larger tables]
```

Both methods will work 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 columns. 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.

4.4.3 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 the short 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.

- `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.

More will come off later, when we start looking at the link tables. The code in listing 4.8 does the trick very simply.

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

```
language="python" linenumbering="unnumbered">logs = logs.drop("BroadcastLogID", "Seq
```

Just like `select()`, `drop()` takes a `*cols` and returns a data frame, this time excluding the columns passed as parameters.

WARNING Unlike `select()`, where selecting a column that doesn't exist will return a runtime error, dropping a non-existent column will return the data frame intact. 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

just what you specify. We could reproduce listing 4.8 with a `select()` method, and listing 4.9 does just that.

Listing 4.9 Getting rid of columns, select-style

```
language="python" linenumbering="unnumbered">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", "SequenceN
    )
```

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[]
```

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

In theory, 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()`, 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()`.

4.4.4 Creating 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: just like 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()`.


```
# +-----+
# |          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
```

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

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

- 1 The original
- 2 `tbl` first two sanity
- 3 checks and their
- 4 `tbl` version have the
- 5 identical second

added a distinct()
to the results

```
# |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
```

The `substr` method takes two parameters. The first gives the position of where the sub-string starts, and the second gives the length of the sub-string we want to extract. We then cast them as `int` using the `cast` method so we can treat them as integers. Casting is covered in more details in Chapter 5. We finally provided an alias for each column so we know easily which one is which.

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 listing 4.11 takes a crack at it.

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

```
language="python" linenumbering="unnumbered">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 our new friend `withColumn()`. Applied to a data frame, it'll return a data frame with the new column appended. Listing 4.12 takes our field and add it to our logs data frame. I've also included a sample of the `printSchema()` so you can see the column added at the end.

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

```
language="python" linenumbering="unnumbered">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)
```

WARNING If you're creating a column with `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 it's the effect you want!

We can create columns using the same expression with `select()` and `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.

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

```
df_grocery_list.withColumn("New Column",[...]).show()
```

Item	New_Column
Banana	
Apple	
Carrot	
Cake	

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

4.4.5 Renaming and re-ordering columns

Renaming columns can be done with `select()` and `alias`, of course. We saw briefly in Chapter 2 that PySpark provides you a easier way to do so. Enter `withColumnRenamed()`! Listing 4.13 will give you a refresher.

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

```
language="python" linenumbering="unnumbered">logs = logs.withColumnRenamed("Duration", "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)
```

©Manning Publications Co. To comment go to liveBook
Licensed to Mike Rumore, #bookjockey@gmail.com

```
language="python"  linenumbering="unnumbered">logs.toDF(*[x.lower() for x in logs.columns])

# 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)
```

Our final step is *re-ordering* columns. Can you guess which method we'll be using?

```
select() !
```

```
language="python"  linenumbering="unnumbered">logs.select(sorted(logs.columns)).  
# 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)
```

① Remember that, in most programming languages, capital letters comes before lowercase ones.

I think that we have exhausted what we can do with our centre table alone. Time to get some of those links to good use!

4.5 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 slide, 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. This is the easiest use case: joins can get dicey, and Chapter 7 will drill deeper into the subject.

If you recall our star schema and how to denormalize data diagrams (figure 4.3 and figure 4.4, respectively), you already have a good idea on how we will proceed. We'll start with one table, explore the process, then generalize it for the other useful tables. The reference tables are under `./data/Ch03/ReferenceTables/`. We read the data using the exact same options of 4.3.

Listing 4.16 Exploring our first link table: `log_identifier`

```
language="python" linenumbering="unnumbered">log_identifier = spark.read.csv(
    "./data/Ch04/ReferenceTables/LogIdentifier.csv",
    sep="|",
    header=True,
    inferSchema=True,
)

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

log_identifier.show(5)
# +-----+-----+-----+
# |LogIdentifierID|LogServiceID|PrimaryFG|
# +-----+-----+-----+
```

① Channel identifier
② Channel key (maps to channel)
③ Boolean flag: is the channel primary (1) or not (0)? We want only the 1's

```

# |          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

```

With our table ingested, onto the join!

A join operation in PySpark (as in many other data processing engines) combines columns from one or more data frames. For this, you need three elements:

1. *Keys*, or columns to join on.
2. *Conditions*, or what is the condition on the keys for the join to be performed
3. *Direction*, or how to treat rows that didn't match in the join operation.

The joins we'll be performing in this Section are called *equi-joins* and are the simplest ones. Basically, you have an identically named columns (keys) in both data frames, and you need to match for equality (condition). Records without a match are dropped (direction). Figure 4.6 shows how it works on a sample of our two data frames. Every record in our logs table that has a LogServiceID which also appears in the log_identifier table will make it to the joined table, combining the matching records from both location. In our example, the LogServiceID=0 doesn't match any record in log_identifier, so the whole row gets dropped.

logs				log_identfier		
LogServiceID	LogDate	...	Duration_seconds	LogIdentifierID	LogServiceID	PrimaryFG
3157	2018-08-01 00:00:00	...	15	13ST	3157	1
3466	2018-08-01 00:15:00	...	60	2000SM	3466	1
3590	2018-08-01 00:45:00	...	45	70SM	3883	1
0	2018-08-01 01:04:00	...	3600	80SM	3590	1
				90SM	3470	1

logs.join(log_identfier, "LogServiceID")					
LogServiceID	LogDate	...	Duration_seconds	LogIdentifierID	PrimaryFG
3157	2018-08-01 00:00:00	...	15	13ST	1
3466	2018-08-01 00:15:00	...	60	2000SM	1
3590	2018-08-01 00:45:00	...	45	80SM	1
0	2018-08-01 01:04:00	...	3600	null	null

Figure 4.6 Equi-join on our two 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 comparison between records, they both need 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 amount of data over the network is very slow, and we should aim to avoid them.

This is one of the instance where PySpark's abstraction model shows some weaknesses. 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. Performance considerations and join strategies will be revisited in Chapter 7. For the moment, trust the optimizer!

4.5.1 The blueprint of a simple join

Whether we're joining two tables together, big or small, the `join` method will look very familiar to you. Listing 4.17 shows how an equi-join would be performed, in a very verbose way.

Listing 4.17 A very verbose and potentially confusing join

```
language="python" linenumbering="unnumbered">logs_and_channels = logs.join(
    log_identfier, logs["LogServiceID"] == log_identfier["LogServiceID"], how="inn
)

logs_and_channels.printSchema()
```



```
# root
# |-- LogServiceID: integer (nullable = true)
# |-- 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)
# |-- PrimaryFG: integer (nullable = true)
```

1 Our original LogServiceID column

2 Another LogServiceID column?

The `join` method takes three parameters, which we introduced conceptually at the beginning of the section.

1. The first one is the table you wish to join on. In our case, we want to join on `log_identifier`.
2. The second one is the condition on which to join. Since we're performing an equi-join, we want the `LogServiceID` to be identical on both tables.
3. The third one is how we want to perform the join. The default is `inner`, which is what we want, but I wanted to make it explicit since it's our first join. Other join types will be encountered in Chapter 5.

This join works fine, but if you were to select the `LogServiceID` column, you'd get an error: PySpark wouldn't know which one you'd want! Let's revisit and simplify our join in listing 4.18 to avoid this problem.

Listing 4.18 A simplified version of our equi-join

```
language="python" linenumbering="unnumbered">logs_and_channels = logs.join(log_iden
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)
```

No more duplicate column! What did we do differently?

When performing a join, if your columns have the same name, passing that

name will automatically match the column on both sides of the table and perform an equality test. We're avoiding some typing and the duplicate column problem all at once! If we were to join on multiple columns, we could simply pass a list of columns.

TIP

If your columns aren't the same name but you want to perform an equi-join, I recommend you rename them before using `withColumnRenamed()` and then perform the join.

We'll conclude this section with the remainder of the links. Two additional links will be necessary to answer our questions.

1. `CategoryID` will be important to link so we can identify the type of programs.
2. `ProgramClassID` will be important so we can identify the commercials.

One key difference from our previous join is that we'll perform *left* joins here. Left refers to the table before the `.join` in our code, and means that all the records of that table will remain, regardless of a match with the right table. Figure 4.7 shows an example of a left join: we can see that, unlike the equi-join, all the records in our `logs` table are present, regardless of a match in our `logs_identifier` table. If there is no match, `null` values are taking place.

logs				log_identifier		
LogServiceID	LogDate	...	Duration_seconds	LogIdentifierID	LogServiceID	PrimaryFG
3157	2018-08-01 00:00:00	...	15	13ST	3157	1
3466	2018-08-01 00:15:00	...	60	2000SM	3466	1
3590	2018-08-01 00:45:00	...	45	70SM	3883	1
0	2018-08-01 01:04:00	...	3600	80SM	3590	1
				90SM	3470	1

logs.join(log_identifier, "LogServiceID", how="left")					
LogServiceID	LogDate	...	Duration_seconds	LogIdentifierID	PrimaryFG
3157	2018-08-01 00:00:00	...	15	13ST	1
3466	2018-08-01 00:15:00	...	60	2000SM	1
3590	2018-08-01 00:45:00	...	45	80SM	1
0	2018-08-01 01:04:00	...	3600	null	null

Figure 4.7 A visual explanation of a left join

Listing 4.19 shows the code to link the two tables we identified, using a left join. The code should look very familiar, see if you can follow along!

Listing 4.19 Linking the category and program class tables using a left join

```
language="python" linenumbering="unnumbered">cd_category = spark.read.csv(
  "./data/Ch04/ReferenceTables/CD_Category.csv",
  sep="|",
  header=True,
  inferSchema=True,
).select(
  "CategoryID",
  "CategoryCD",
  F.col("EnglishDescription").alias("Category_EnglishDescription"),
)

cd_program_class = spark.read.csv(
  "./data/Ch04/ReferenceTables/CD_ProgramClass.csv",
  sep="|",
  header=True,
  inferSchema=True,
).select(
  "ProgramClassID",
  "ProgramClassCD",
  F.col("EnglishDescription").alias("ProgramClass_EnglishDescription"),
)

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

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

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

This section described very succinctly how to perform the two most common types of join: the equi-inner-join and the equi-left-join. From my experience, those two will make up more than 80% of the joins you'll be using when working with tabular or normalized data. The syntax is quite easy to remember:

- The `left` table is the one before (or to the left of) the `.join()`
- The `.join()` method takes as a first parameter the `right` table
- The second parameter is the match test. You can either provide a boolean test (`left["column"] == right["column"]`) or, if the column has the same name in both tables, just the column name as a string (`"column"`).
- Finally, the `how` parameter specifies the type of join. We've learned `inner` and `left`, which performs an inner and a left join respectively. Omitting this parameter will default to an inner join.

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

4.6 Summarizing the data via: groupby and GroupedData

Grouping similar records together to compute summaries and statistics is one of the most common operations to display data. We've seen some examples with `summary()` and `display()` which compute mean, min, max, etc. over the whole data frame. What if we need to look using a different lens?

For our example, we need to summarize it in a way 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 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!

4.6.1 A simple groupby blueprint

In Chapter 2, we performed a very simple group by to count the occurrences of each word. We'll expand on that simple example by grouping over many columns and using a more general notation than the `count()` we've used. Listing 4.20 tells all about it.

Listing 4.20 Displaying the most popular types of programs

```
language="python" linenumbering="unnumbered">full_log.groupby("ProgramClassCD", "Pr
    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

1. If you want to add another aggregate function, you can add as an additional parameter to `.agg()`. 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 nomenclature and improve the robustness of your code.

```
logs.groupby("LogServiceID"): GroupedData
```

```
logs.groupby(
    "LogServiceID"
).agg(
    F.sum(F.col("Duration_seconds"))
): DataFrame
```

LogServiceID		Duration_seconds
3157	...	15
	...	3600
	...	30
	...	7200
3466	...	60
	...	60
	...	540
3590	...	45

LogServiceID	...	Duration_seconds
3157	...	15 + 3600 + 30 + 7200 = 18045
3466	...	60 + 60 + 540 = 660
3590	...	45

Figure 4.9 My take on a realized `GroupedData` (which is a data frame)

Once we're back in data frame-land, we can use the `orderBy` method to order the data by decreasing order of `duration_total`, our newly created column.

We end up showing 100 rows, which is more than what the data frame contains, so it shows everything.

Let's select our commercials. 4.1 shows my picks.

Table 4.1mThe 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!

4.6.2 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 4.1 into a PySpark definition, this gives us listing 4.21

Listing 4.21 Computing only the commercial time for each program in our table

```
language="python" linenumbering="unnumbered">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 follow. 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], [RESULT IF TRUE])
  .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. XREF ch04-commercial does just that, and at the same time, gives us our answer!

Cache & Persist: Saving your progress 4 Using our new column into `.agg()` to compute our final answer

The final aspect of data semantic is understanding which of our transformations are worth keeping and which one are transitional. By telling our program what to save to memory (or to disk), you can speed up your computations and avoid potential out-of-memory errors.

A PySpark data frame is like those old NES games, where your data is the game and you're giving commands to your character. Here, lives aren't at stake, but if you crash your program in the middle of Step 100 of 250, PySpark will crash the whole chain of computations and you'll have to recompute everything from the start.

Now, since the introduction of the NES gaming console, game developers listened to our qualms and introduced a *save* (or checkpoint) feature so you don't have to start from the start when you're game over. PySpark provides a similar functionality through the `cache()` method.

How caching works

When talking about a data frame, we rightfully focus a lot about the data and how it is being organized. In Chapter 1, we saw that this is only part of the power of the data frame. By leveraging lazy computation and storing them until processing is absolutely necessary, PySpark can further optimize our transformations. We will call those stored operations the "logical plan" of a PySpark data frame.

Looking at logical plans in PySpark is something we will worry when we start looking deeper into PySpark performance, but we can show a quick example to demonstrate how caching works in practice. In XREF ch05-code-caching-example, we are taking a small data set, performing a few transformations and then using the `explain()` method to display the logical plan. At the moment, our data frame looks somehow like XREF ch05-figure-pre-cache. We have our data container displayed on the right, with its associated physical plan.

When we call `cache()` in the REPL, it returns immediately, since it's considered by PySpark to be a transformation. It sends the directive for PySpark to cache the data frame at its earliest convenience, but doesn't perform any work. The data frame will be cached when we submit an action.

Balancing caching and processing

Caching is incredibly useful and powerful, yet very misunderstood. Beginner Spark developers will tend to over-cache because they fear to make a mistake that will send them back a couple dozen lines of code. This can be compounded by the tendency to build those massive pipelines that do too much at once and are hard to debug.

```
full_log.groupby("LogIdentifierID").agg( F.sum( F.when( F.trim(F.col("ProgramClassCD")).isin( ["CC", "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") ).orderBy( "commercial_ratio" ascending=False ).show( 1000, False ) # +-----+-----+-----+-----+
|LogIdentifierID|duration_commercial|duration_total|commercial_ratio | #
+-----+-----+-----+-----+ # |HPITV |403 |403 |1.0 | # |TLNSP |2
|234455 |1.0 | # |MSET |101670 |101670 |1.0 | # |TELENO |545255 |545255 |1.0 | # |CIMT |19935 |19935
|TANG |271468 |271468 |1.0 | # |INVST |623057 |633659 |0.9832686034602207 | # [...] # |OTN3 |0 |26
|PENT |0 |2678400 |0.0 | # |ATN14 |0 |2678400 |0.0 | # |ATN11 |0 |2678400 |0.0 | # |ZOOM |0 |2678400
|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 they don't broadcast a lot were in our

data set (1 day = 86,400 seconds). There is something puzzling in our data: could it be possible that our data isn't as pristine as we thought? Still, we accomplished our goal: we identified the channels with the most commercials. Data tidying will be a job for Chapter 5 us, and not before!³

Footnote 3 But if you're interested in sharpening your skills and uncover some potential avenues for improving our analysis, have a look at the exercises.

4.7 What was our question again: our end-to-end program

Our goal was to identify the channels with the most and least commercials in their programming. Let's assemble our interactively written code in a complete and cohesive package. Listing 4.23 showcases all the code, including the PySpark header boilerplate. You'll be able to see the code also in the repository, under `./code/Ch03/commercials.py`.

Not counting data ingestion, 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 about 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 concludes the "walking" part of our journey. We created two simple data programs on structured (tabular) and unstructured (textual) data. You should now be able to take a problem statement and, using the material you've learned so far (and perhaps a little bit of help from the `pyspark` shell, like we did in Chapter 2), create your own data manipulation programs. Appendix D will also give you a good walk-through the PySpark online API, which is an incredible tool to master once you start writing your own programs.

The second part of the book, "run" will focus on more advanced subjects that you might encounter a little less often in your PySparkian life.

- You'll learn how to seamlessly blend Python and SQL code within a PySpark program, which will make moving from relational databases (and their huge body of SQL code) much easier.
- You'll learn the power of PySpark's types, the many pitfalls they contain, and how to avoid them.
- You'll learn how to use PySpark to ingest, make sense of, and tidy messy or poorly formatted data.
- Finally, you'll go beyond rows and columns by using the document-data capacities of PySpark.

Like they say, *you haven't seen nothing*. See you in the next section!

Listing 4.23 Our full program, ordering channels by decreasing proportion of commercials in their airings.

```
language="python" linenumbering="unnumbered">import os
from pyspark.sql import SparkSession
import pyspark.sql.functions as F

spark = SparkSession.builder.appName(
    "Getting the Canadian TV channels with the highest and lowest proportion of comm
").getOrCreate()

spark.sparkContext.setLogLevel("WARN")

#####
# Reading all the relevant data sources
#####

DIRECTORY = "./data/Ch03"

logs = spark.read.csv(
    os.path.join(DIRECTORY, "BroadcastLogs_2018_Q3_M8.CSV"),
    sep="|",
    header=True,
    inferSchema=True,
)

log_identifier = spark.read.csv(
    "./data/Ch03/ReferenceTables/LogIdentifier.csv",
    sep="|",
    header=True,
    inferSchema=True,
)

cd_category = spark.read.csv(
    "./data/Ch03/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"
)

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

```

4.8 Summary

- Reading delimited data in PySpark can be done by using the `spark.read.csv()` function. Many options are available to account for the variety of formatting the most popular are separators, schema and presence of header.
- PySpark's data frame are very well suited for tabular or relational data. Each column is, literally, a column.
- You've learned to use the two methods for summarizing data, `.describe()` and `.summary()`.
- You have learned how to select, manipulate, rename and delete columns in a data frame using its methods. Furthermore, you also modified columns by using its methods, but also by using the vast array of functions available in `pyspark.sql.functions`.
- You've joined two data frames together using equality testing (equi-join, equi-left-join) via the `.join()` method.
- You've grouped a data frame according to columns' values and performed aggregate functions on the result to distill information.

4.9 Exercises

[Practical] Exercise 4.1

Using the data from the `data/Ch04/Call_Signs.csv` (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.

[Practical, Beyond] Exercise 4.2

Many channels don't seem to be broadcast long enough to be significant. How would you filter the data frame to only keep the channels that broadcast at least more than the average of all the channels?

[Conceptual/Practical] Exercise 4.3

Reproduce the results from Listing 4.7 using the `select()` statement.

[Practical] Exercise 4.4

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.

[Practical] Exercise 4.5

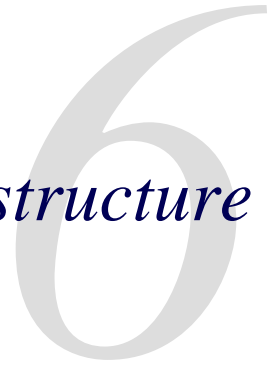
On the data frame returned from `commercials.py`, return the percentage of channels in each bucket based on their `commercial_ratio`. (Hint: look at the documentation for `round` in how to truncate a value.)

commercial_ratio	proportion_of_channels
1.0	
0.9	
0.8	
...	
0.1	
0.0	

[Conceptual, Beyond] Exercise 4.6

We saw simple joins in this Chapter. In your opinion, and with what we've learned about Spark so far (you can refer to Chapter 1 if necessary), why are joins treated so gingerly in Spark? What problems can you foresee as the data size grows?

Making sense of your data: types, structure and semantic



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 through types, how they can guide our operations and how to avoid common mistakes when working with them.

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 is 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, 6.8 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 word 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 6.8. 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 6.5). Let's take a very simple data frame to demonstrate four different categories of types. The ingestion and schema display happens in 6.1. 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) ①
# |-- integer_column: integer (nullable = true) ②
# |-- float_column: double (nullable = true) ③
# |-- date_column: timestamp (nullable = true) ④
```

- ① string_column was inferred as being a string type (StringType(), 6.1)
- ② integer_column was inferred as being an integer type` (IntegerType(), 6.2)
- ③ float_column was inferred as being a double type (DoubleType(), 6.3)
- ④ date_column was inferred as being a timestamp type (TimestampType() 6.4)

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 6.1 and 6.1 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 6.1, 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 6.1 to 6.5.

Table 6.1 A summary of the types in PySpark. A star means there are loss of precision.

Type Constructor	String representation	Python equivalent
<code>NullType()</code>	null	None
<code>StringType()</code>	string	Python's regular strings
<code>BinaryType()</code>	N/A	bytearray
<code>BooleanType()</code>	boolean	bool
<code>DateType()</code>	date	datetime.date (from the datetime library)
<code>TimestampType()</code>	timestamp	datetime.datetime (from the datetime library)
<code>DecimalType(p,s)</code>	decimal	decimal.Decimal (from the decimal library)*
<code>DoubleType()</code>	double	float
<code>FloatType()</code>	float	float*
<code>ByteType()</code>	byte OR tinyint	int*
<code>IntegerType()</code>	int	int*
<code>LongType()</code>	long OR bigint	int*
<code>ShortType()</code>	short OR smallint	int*
<code>ArrayType(T)</code>	N/A	list, tuple or Numpy array (from the numpy library)
<code>MapType(K, V)</code>	N/A	dict
<code>StructType(...)</code>	N/A	list or tuple

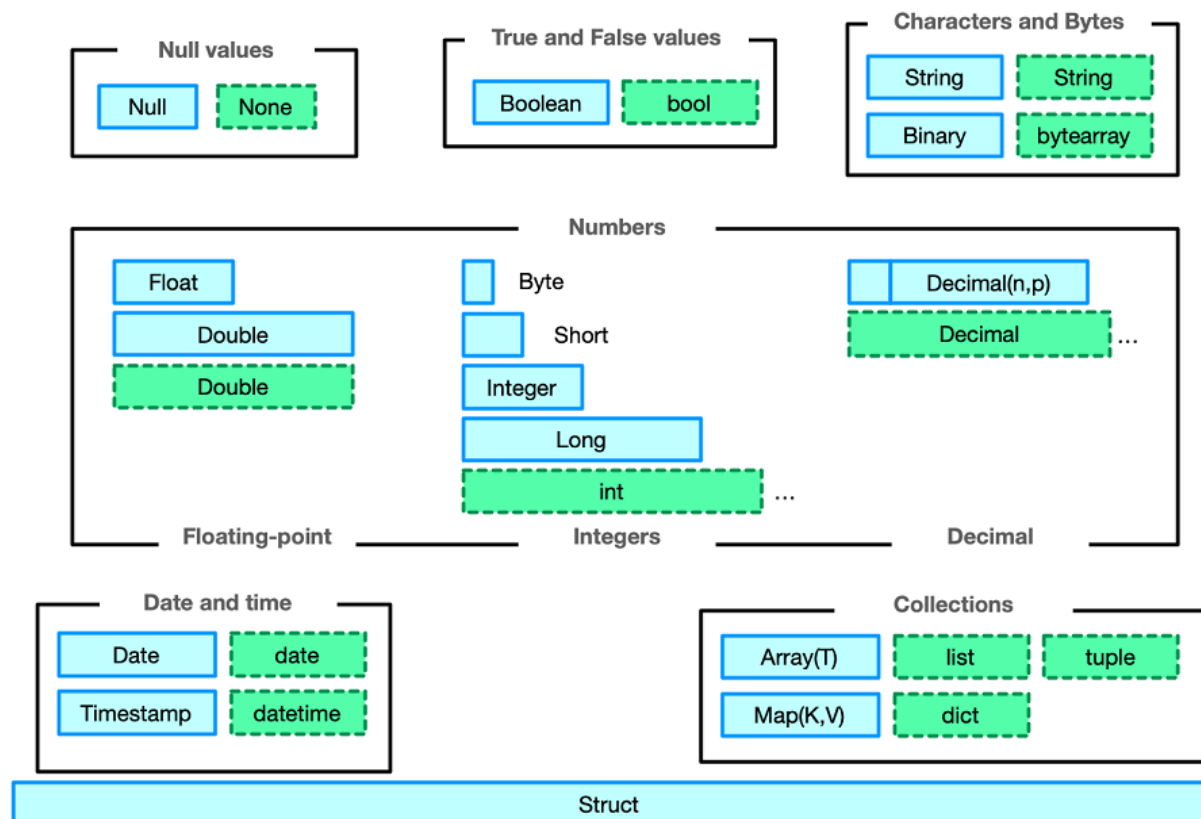


Figure 6.1 An illustrated summary of PySpark's types taxonomy, with Python equivalents

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 6.4), or numerical values ([ch06-numerical-integer](#)). 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.

#	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]

- 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. 6.2 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. 6.3 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
# +-----+-----+-----+
```


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 6.4 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` (6.3) will allow for larger numbers. We will also see in 6.8 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. 6.5 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 6.6. 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|
# +-----+-----+-----+
```


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 (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 (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 wish 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 (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 6.8, 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. 6.3 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:mm") 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.

- ④ If you don't, PySpark will silently default them to `null` values.

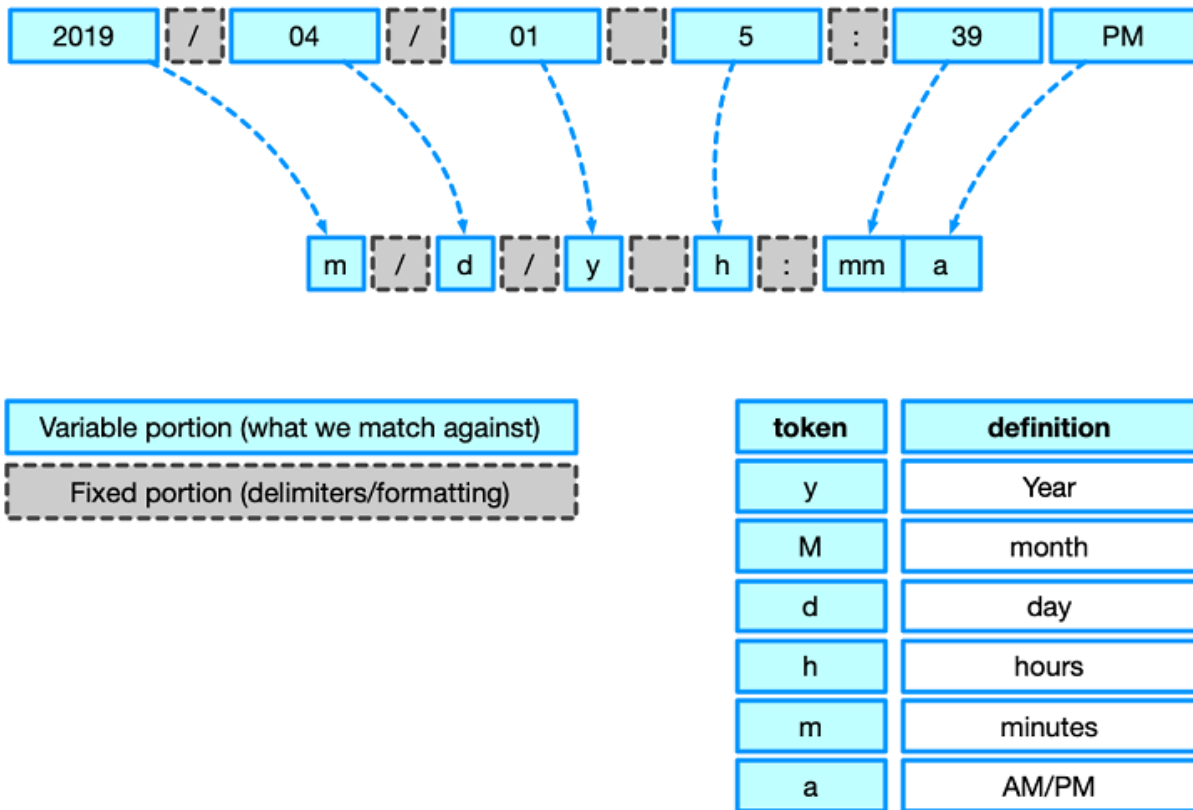


Figure 6.3 Parsing 04/01/2019 5:39PM as a timestamp object using PySpark/Java's SimpleDateFormat's grammar

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 6.11, 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 6.8.

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())`. 6.12 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, so the resulting type is `Array(Long)`
- ❸ An array can contain null values. Of course, reading a 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 Bulbapedia⁶, 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* (Pokemon with two types) in the Pokedex?

The code in 6.13 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 Pokemon 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.

©Manning Publications Co. To comment go to liveBook
Licensed to Mike Rumore, Mike Rumore <mike.rumore@gmail.com>

- 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 6.7 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 6.15, 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.

©Manning Publications Co. To comment go to liveBook
Licensed to Mike Rumore, #bookjockey@gmail.com

```
# 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 6.10) 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 6.1 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` (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 6.16. 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 to, 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 [ch06-numerical-integer](#).

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 6.18 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|2019/12/25|null|
# +-----+-----+-----+
```

- ❶ Casting our string-encoded decimal numbers into a double
- ❷ Casting our string-encoded dates into a date

- ③ As a simple example, we split the strings at the underscore character and keep the group before. This will take care 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 6.4.

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 6.20, 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 6.18.
- ❷ PySpark will use the same type promotion for arithmetic values we've seen in 6.2. 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

Exercise 5.1

How could you demonstrate that PySpark stores `Date` as `Timestamp` with the time forced to 00:00:00?

Exercise 5.2

Taking the `cast_df` from 6.20, how could you fill the null values with the date 1900-01-01?

Appendix A: Installing PySpark locally

SIDEBAR

Last update: 2019-09-25

Python 2 is on the downfall, with a scheduled end-of-life on January 1st, 2020. The transition will happen while this book is in writing. 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

- Windows 10
- OS.X Mojave
- Ubuntu 18.04 LTS

As OS version will get updated, I'll continue updating this Appendix.

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

We will use the command line as much as possible, to make the installation steps easily reproducible. We are covering the following options:

- Windows 7 and 10 (plain installation)
- Windows 10 with Windows Subsystem for Linux (WSL)
- Mac Os.X Mojave, using Homebrew
- Linux (Ubuntu), using apt

NOTE

You'll need admin rights on the machine you're trying to install Spark on. I find that fiddling to make it work on a locked-down work computer is often not worth it. If you can't install it using the instructions in this Appendix, have a look at low-cost/no-cost cloud options in Appendix B.

A.1 Preliminary steps

Depending on your OS and installation strategy, there are some OS specific steps we need to follow. Table A.1 lists them: please have a look to see if your OS is listed.

Table A.1 Preliminary steps to accomplish before installing PySpark on your personal computer

OS	Preliminary Steps	Section
Windows (plain)	Install 7-zip	Appendix A.1.1
Windows (WSL)	Install WSL	Appendix A.1.2
Os.X	Install HomeBrew	Appendix A.1.3

A.1.1 Windows (plain): 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-zip⁷. Simply go on the website, download the program and follow the installation instructions.

A.1.2 Windows (WSL): Install WSL

Windows Subsystem for Linux is similar to a Linux virtual machine running on your windows OS. It's easy to install, configure, and integrates seamlessly with your Windows. When using Windows, I use Spark via the WSL because I find it simpler to install.

The first step is to make sure the WSL flag is enabled in your Windows installation. Go to aka.ms/wslinstall and follow the instructions on the website. You'll be asked to reboot after.

Once your computer is rebooted, search for "Ubuntu" in the Windows Store. This will install Ubuntu as a WSL distribution. You can now follow the Linux/Ubuntu instructions to install PySpark!

A.1.3 OS.X: 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.

A.2 Step 1: Install Java

Spark works best using Java 8. You might already have Java on your computer. On Windows, look for the "Java" and "JRE" keywords in the list of installed programs. You can also (all OS) open a terminal and type the following.

```
java -version
```

Look for something looking like "version 1.8.0_XYZ". If you have 1.11 or 1.12, it should work as well. The Spark website provides a compatibility matrix with Scala/JVM versions on the official documentation which evolves rather quickly.

A.2.1 Windows (plain)

The easiest way to install Java on Windows is to go on www.java.com and follow the download and installation instructions. Make sure to read the installer steps to avoid installing non-useful software!

A.2.2 Os.X

With Homebrew installed, open a terminal and type the following

```
$ brew cask install homebrew/cask-versions/adoptopenjdk8
```

The installer will prompt for your password during the installation process.

A.2.3 GNU/Linux Ubuntu and WSL

Most GNU/Linux distributions provide a package manager. OpenJDK version 8 is available through the software repository.

```
`sudo apt-get install openjdk-8-jre`
```

Now that Java is installed, we can now look at installing Spark.

A.3 Step 2: Installing Spark

Spark is available through the Apache Project website (spark.apache.org). The instructions are pretty much identical for every OS beside Os.X, since Homebrew provides Spark as a package.

A.3.1 Windows (plain), GNU/Linux (including WSL)

Go on the Apache website and download the latest Spark release. You shouldn't have to change the default options, but A.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).

Download Apache Spark™

1. Choose a Spark release:
2. Choose a package type:
3. Download Spark: [spark-2.4.4-bin-hadoop2.7.tgz](#)
4. Verify this release using the 2.4.4 [signatures](#), [checksums](#) and [project release KEYS](#).

Note that, Spark is pre-built with Scala 2.11 except version 2.4.2, which is pre-built with Scala 2.12.

Figure A.1 The options to download Spark

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 Shotts⁸. 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 or move the directory to your liking. I'll keep it where it is for the remainder of the Appendix.

If you are on GNU Linux or WSL, you can skip to the next section. **For plain Windows installation**, you'll also need to download a file called `winutils.exe` and set a few environment variables to prevent some cryptic Hadoop errors. Go on the github.com/cdarlint/winutils

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.



Homebrew strikes again! Input the following command in a terminal.

©Manning Publications Co. To comment go to liveBook
Licensed to Mike Rumore, mikerumore@gmail.com

A.4 Step 3: Install Python 3 and IPython

On Windows, Python doesn't come installed. OS.X provides Python 2. Ubuntu provides no `python` command, preferring the `python2` and `python3` commands. We'll provide a surefire way to get Python 3 for Windows and OS.X, and provide a guide to setup the default Python 3 on Ubuntu with IPython.

A.4.1 Windows, Os.X

The easiest way to get Python 3 is to use the Anaconda Distribution. Go on 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 inputting in a command line.

```
$ conda activate base
```

This will prepend a little `(base)` to your shell prompt, meaning that you're now working within the base anaconda python environment.

TIP

On Windows, you'll need to use the "Anaconda Powershell Prompt" that you can find in your Start Menu. By default, `(base)` will be selected for you.

A.4.2 GNU/Linux Ubuntu and WSL

Python 3 is already provided, you just have to install IPython. Input the following command in a terminal.

```
sudo apt-get install ipython3
```

A.5 Step 4: Launch PySpark with IPython

A.5.1 Windows (plain)

Launch the Anaconda Powershell Prompt using the start menu and navigate to the `bin` directory of your Spark installation. (You can use the `cd` command, which stands for "Change Directory" to move around.

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).

In order to use IPython as a front-end for PySpark, you have to set the proper environment

variable. Use the following code block in o

```
Set-Item Env:PYSARK_DRIVER_PYTHON ipython
pyspark.cmd
```

A.5.2 Os.X

Assuming that the terminal points to the `bin/` folder of where Spark was unzipped, you just have to use the following command.

```
`PYSARK_DRIVER_PYTHON=ipython3 PYSARK_PYTHON=python3 pyspark`
```

PYSARK_DRIVER_PYTHON will give you the shell flavor to use on the driver we interact with.

NOTE

If you launch PySpark and see Python version **2.7** instead of **3.X**, it means you haven't `conda activate base` before launching PySpark. `exit` the PySpark shell and have a look at A.4.1.

A.5.3 GNU/Linux Ubuntu and WSL

Since Ubuntu doesn't provide a plain `python` command anymore, we have to specify two environment variables to use PySpark with IPython and Python 3. Assuming that the terminal points to the `bin/` folder of where Spark was unzipped, you just have to use the following command.

```
PYSARK_DRIVER_PYTHON=ipython3 PYSARK_PYTHON=python3 pyspark
```

PYSARK_PYTHON gives Spark the indication to use a specific version of Python on the cluster (even if we're on a local machien), while PYSARK_DRIVER_PYTHON will give you the shell flavor to use on the driver we interact with.

Notes

- It can be a fun probability exercise to compute by how much, but I will try to keep the math stuff at a
1. minimum.

- Application Programming Interface, which is basically the set of functions, classes and variables provided
2. for you to interact with

- Java Virtual Machine, which is like an emulator running on your computer. Both Java and Scala targets the
3. 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. bulbapedia.bulbagarden.net/wiki/List_of_Pok%C3%A9mon_by_National_Pok%C3%A9dex_number

7. www.7-zip.org/

8. linuxcommand.org/