Introduction to `pyspark`

Pedro Duarte Faria

2022-04-24

Table of contents

Welcome! **4**

Preface **5**
Introduction . . . 5
Some conventions of this book . . . 5
Python code and terminal commands . . . 5
Python objects, functions and methods . . . 6
Be aware of differences between OS's! . . . 6
Opening the terminal of your OS . . . 7
Opening a terminal on Windows . . . 7
Opening a terminal on Linux . . . 9
Opening a terminal in MacOs . . . 9
First step: install the necessary software . . . 11
Install python . . . 11
Install `pyspark` . . . 11
Install `Spark` . . . 12

1 Key concepts of python **13**
1.1 Introduction . . . 13
1.2 Scripts . . . 13
1.3 How to run a python program . . . 13
1.4 Objects . . . 14
1.5 Expressions . . . 16
1.6 Packages . . . 18
1.7 Methods versus Functions . . . 21
1.8 Identifying classes and their methods . . . 22

2 Introducing Apache Spark **24**
2.1 Introduction . . . 24
2.2 What is Spark? . . . 24
2.3 Spark application . . . 25
2.4 Starting your Spark Session . . . 26
2.5 Running your first Spark application . . . 27

3 Introducing Spark DataFrames **29**
3.1 Introduction . . . 29

3.2 Spark DataFrames versus Spark Datasets . . . 29
3.3 Building a Spark DataFrame . . . 31
3.4 Viewing a Spark DataFrame . . . 33
3.5 Spark Data Types . . . 34
3.6 The DataFrame Schema . . . 35
3.6.1 Accessing the DataFrame schema . . . 35
3.6.2 Building a DataFrame schema . . . 37
3.6.3 Checking your DataFrame schema . . . 38
3.7 The `Column` class . . . 39
3.8 Partitions of a Spark DataFrame . . . 39

4 Transforming your Spark DataFrame 40
4.1 Introduction . . . 40
4.2 Defining transformations . . . 40
4.3 Triggering calculations with actions . . . 42
4.4 Understanding narrow and wide transformations . . . 44
4.5 Filtering rows of your DataFrame . . . 45
4.5.1 Logical operators available . . . 49
4.5.2 Connecting multiple logical expressions . . . 50
4.5.3 Translating the `in` keyword to the pythonic way . . . 54
4.5.4 Negating logical conditions . . . 54
4.5.5 Filtering `null` values . . . 56
4.6 Selecting specific columns of your DataFrame . . . 59
4.6.1 Renaming your columns . . . 60
4.6.2 Dropping unnecessary columns . . . 61
4.6.3 You can add new columns with `select()` . . . 61
4.6.4 Casting columns to a different data type . . . 62
4.7 Calculating or adding new columns to your DataFrame . . . 63
4.8 Sorting rows of your DataFrame . . . 64

5 Working with SQL in Spark DataFrames 67

6 Importing data to Spark 68
6.1 Importing data from different data sources . . . 68
6.1.1 Reading data from static files . . . 68
6.1.2 Defining import options . . . 70
6.1.3 Import options for CSV files . . . 71
6.1.4 Pulling data from SQL Databases . . . 71

References 72

Welcome!

Hello! This is the initial page! This book provides an introduction to `pyspark`, but, because `pyspark` is a python API to Apache Spark, this book is about Apache Spark too.

Preface

Introduction

In essence, `pyspark` is a python package that provides an API for Apache Spark. In other words, with `pyspark` you are able to use the python language to write Spark applications and run them on a Spark cluster in a scalable and elegant way. This book focus on teaching the fundamentals of `pyspark`, and how to use it for big data analysis.

This book, also contains a small introduction to key python concepts that are important to understand how `pyspark` is organized and how it works in practice, and, since we will be using Spark under the hood, is very important to understand a little bit of how Spark works, so, we provide a small introduction to Spark as well.

Big part of the knowledge exposed here is extracted from a lot of practical experience of the author, working with `pyspark` to analyze big data at platforms such as Databricks[1].

Some conventions of this book

Python code and terminal commands

This book is about `pyspark`, which is a python package. As a result, we will be exposing a lot of python code across the entire book. Examples of python code, are always shown inside a gray rectangle, like this example below.

Every visible result that this python code produce, will be shown outside of the gray rectangle, just below the command that produced that visible result. Besides that, every line of result will always be written in plain black. So in the example below, the value `729` is the only visible result of this python code, and, the statement `print(y)` is the command that triggered this visible result.

```
x = 3
y = 9 ** x
```

[1] https://databricks.com/

```
print(y)
```

```
729
```

Furthermore, all terminal commands that we expose in this book, will always be: pre-fixed by `Terminal$`; written in black; and, not outlined by a gray rectangle. In the example below, we can easily detect that this command `pip install jupyter` should be inserted in the terminal of the OS (whatever is the terminal that your OS uses), and not in the python interpreter, because this command is prefixed with `Terminal$`.

```
Terminal$ pip install jupyter
```

Some terminal commands may produce visible results as well. In that case, these results will be right below the respective command, and will not be pre-fixed with `Terminal$`. For example, we can see below that the command `echo "Hello!"` produces the result `"Hello!"`.

```
Terminal$ echo "Hello!"
"Hello!"
```

Python objects, functions and methods

When I refer to some python object, function, method or package, I will use a monospaced font. In other words, if I have a python object called "name", and, I am describing this object, I will use `name` in the paragraph, and not "name". The same logic applies to functions, methods and package names.

Be aware of differences between OS's!

Spark is available for all three main operational systems (or OS's) used in the world (Windows, MacOs and Linux). I will use constantly the word OS as an abbreviation to "operational system".

The snippets of python code shown throughout this book should just run correctly no matter which one of the three OS's you are using. In other words, the python code snippets are made to be portable. So you can just copy and paste them to your computer, no matter which OS you are using.

But, at some points, I may need to show you some terminal commands that are OS specific, and are not easily portable. For example, Linux have a package manager, but Windows does not have one. This means that, if you are on Linux, you will need to use some terminal commands

to install some necessary programs (like python). In contrast, if you are on Windows, you will generally download executable files (`.exe`) that make this installation for you.

In cases like this, I will always point out the specific OS of each one of the commands, or, I will describe the necessary steps to be made on each one the OS's. Just be aware that these differences exists between the OS's.

Opening the terminal of your OS

Every OS comes with a terminal (or command prompt), and you will find this tool very useful for a number of things. I will expose many terminal commands in this book, and these commands should be used inside these terminals that comes with your OS.

The next sub-sections will show you how to open a terminal in each one of the OS's where Spark is available. Lets begin with Windows.

Opening a terminal on Windows

There are some different approaches to do this, but, the one that I find the most useful, is to open a terminal from inside a Windows folder, using the default File Explorer program of Windows. When we use the terminal, we usually want to affect or use a file that is stored inside a folder in our computer, with a terminal command.

Because of that, when we open a terminal from inside a folder, the terminal opened is already rooted inside the folder where our file is stored. In other words, we already have easy access to the file that we want to affect/use in our command, and we do not have the work to change or adjust directories in this terminal.

For example, lets suppose that we want to use a terminal command to use a file called `hello.py`, and, that this `hello.py` file is stored inside a folder called `HelloPython`. You can see in Figure 1, that this folder is in path `C:\Users\pedro\Documents\HelloPython`. So, the first step, is to use the File Explorer of Windows, to open this folder, like in Figure 1.

After you opened this folder, substitute the path to the folder in the search box with the word "cmd", like in Figure 2, and them, press Enter in the keyboard.

As a result, a new terminal will open. See in Figure 3, that this new terminal is already looking to (or is already rooted on) this `HelloPython` folder. This way, we can easily access the files stored inside this folder, like the `hello.py` file.

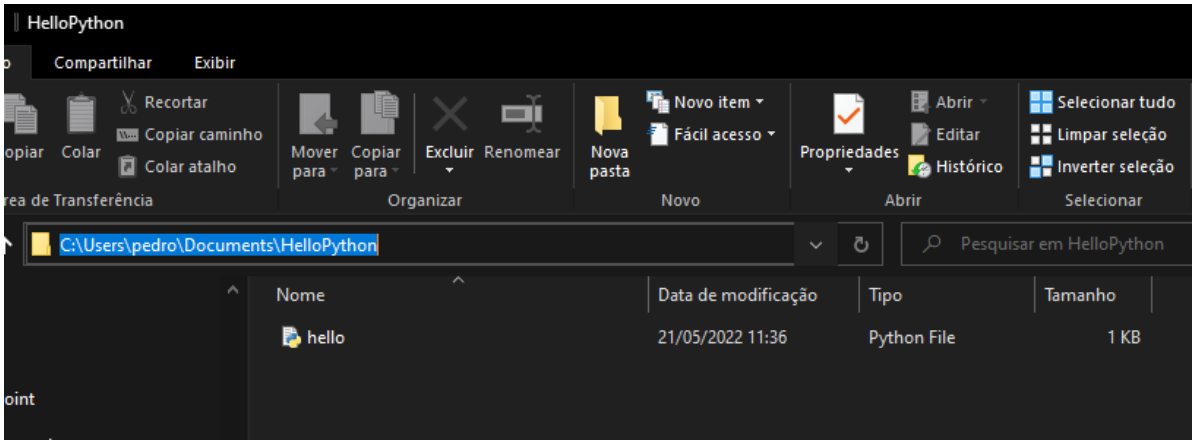

Figure 1: Opening the `HelloPython` folder in Windows

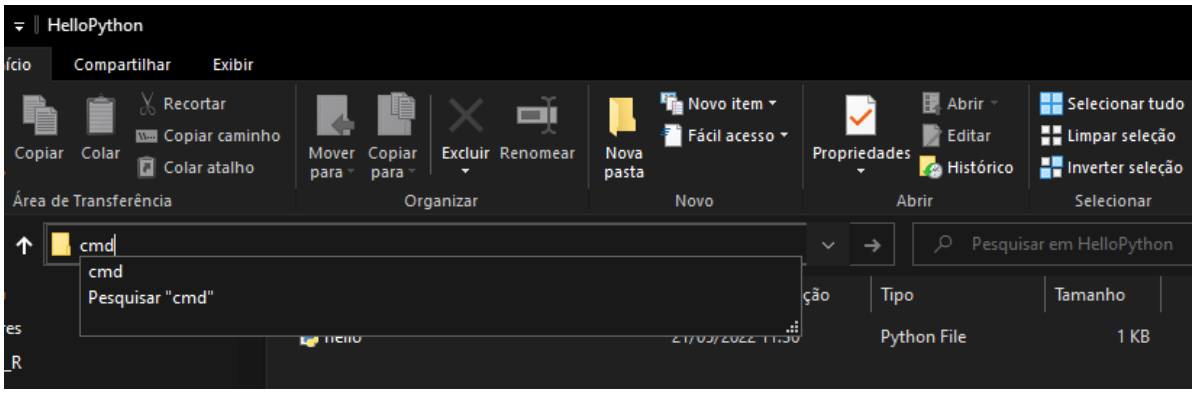

Figure 2: Opening a terminal inside a Windows folder - Part 1

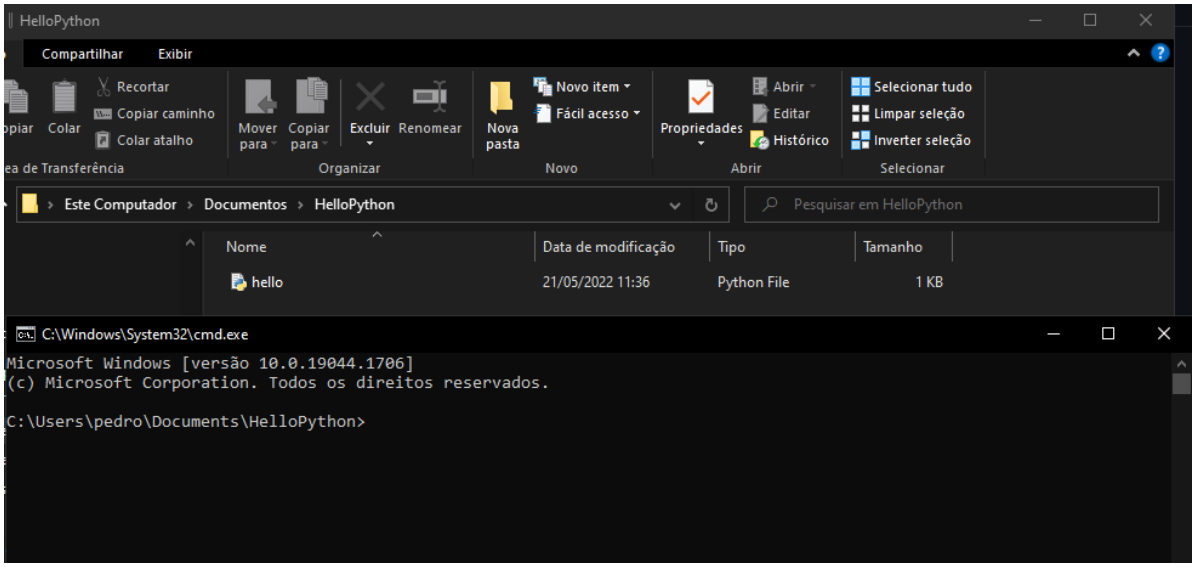

Figure 3: Opening a terminal inside a Windows folder - Part 2

Opening a terminal on Linux

Is fairly easy to open a terminal on a Linux distribution. Again, is very useful when you open the terminal from inside the folder you are interested in. Because you will have an easier access to all the files that are stored inside this folder.

To do this in Linux, you use the built-in File Explorer to open the folder where you want to root your terminal. At the moment, I am using an Ubuntu distribution. I just opened the same `HelloPython` folder, with the same `hello.py` file, in the File Explorer of Linux. As shown in Figure 4:

After you opened the folder, just click with the right button of your mouse, and select the "Open in Terminal" option, and a new terminal should appear on your screen. See in Figure 5, that the new terminal is already looking to the `HelloPython` folder, as we expected.

Opening a terminal in MacOs

Unfortunately, I do not have a Mac machine in my possession, so I cannot easily show you how to open a terminal in MacOs. But there a lot of articles available in the internet discussing how to open a terminal in MacOs. For example, there is a article from the support of Apple[2], or this other article from iDownloadBlog[3].

[2]https://support.apple.com/en-ie/guide/terminal/apd5265185d-f365-44cb-8b09-71a064a42125/mac
[3]https://www.idownloadblog.com/2019/04/19/ways-open-terminal-mac/

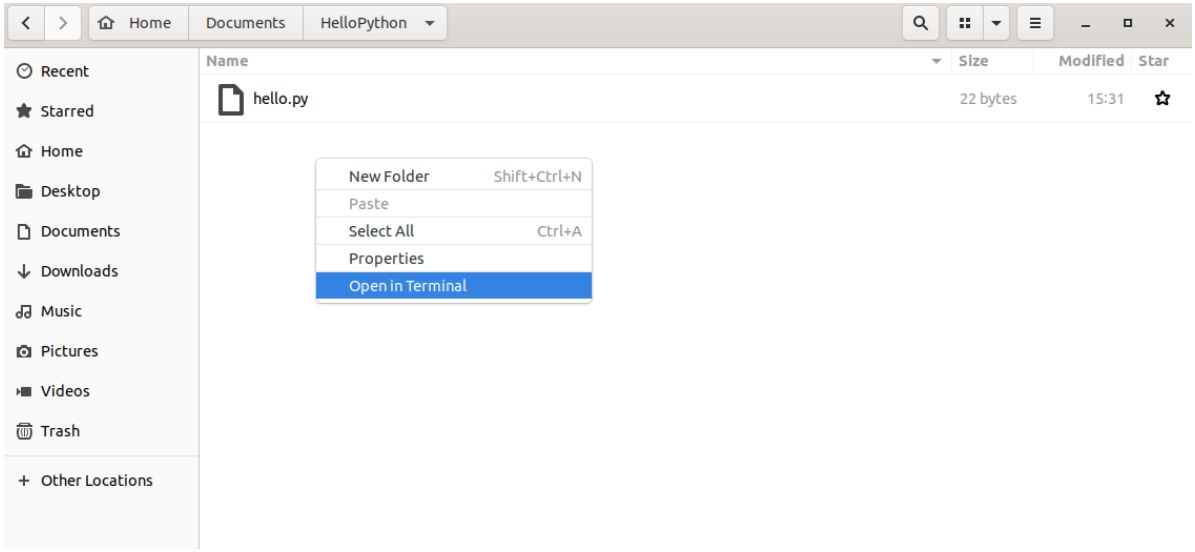

Figure 4: Opening the `HelloPython` folder in Linux

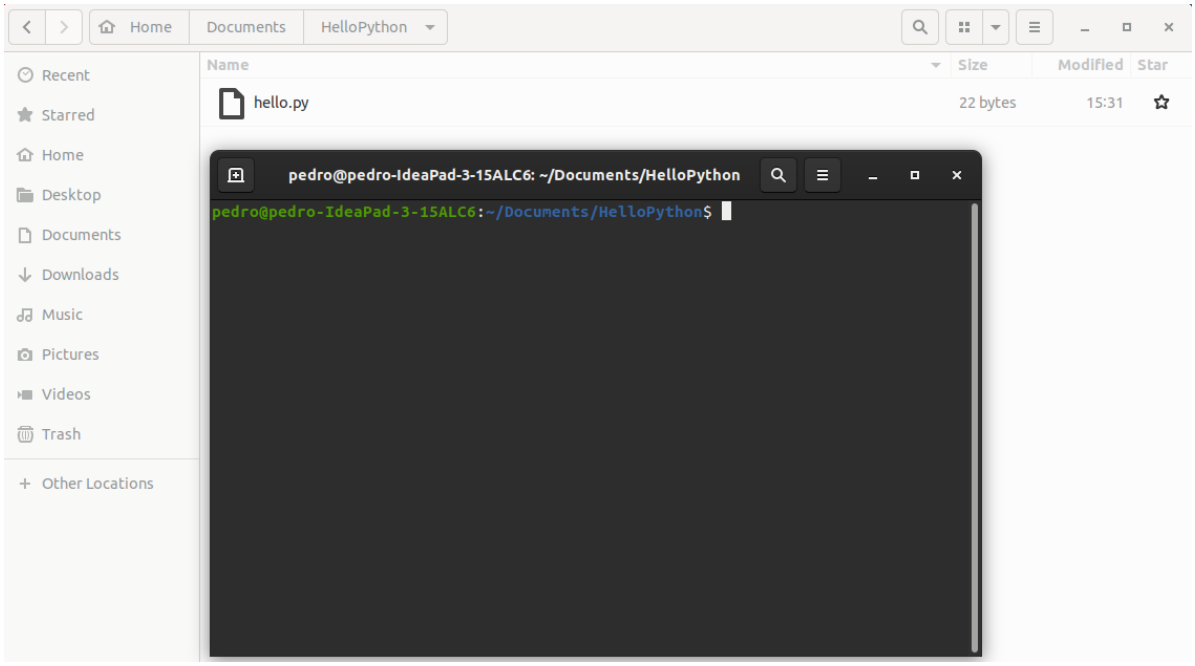

Figure 5: Opening the terminal in Linux

First step: install the necessary software

Before we start, is really important that you have all the software you need already installed on your machine and ready to go. `Spark` and, as a result, `pyspark`, are both available for Windows and UNIX-like systems (e.g. Linux and MacOs). To use `pyspark` you need to install: 1) `Spark`; 2) Python; 3) and the python package called `pyspark`.

Install and manage `Spark` in Windows is a little harder than in a UNIX-like machine. But it's perfectly ok if you do not have a UNIX machine. A Windows machine will do it just fine. Just for referencing, the examples showed throughout this book were executed in a Linux machine (more specifically, in Ubuntu).

Install python

To install python on Windows, you can download the executable files at the official Python website[4]. But, if you are on Linux, you can install python with the package manager, by opening a terminal and running the following command:

```
sudo apt-get install python3
```

Install `pyspark`

Installing the `pyspark` python package is pretty straightforward. Just open a terminal (if you need help to open the terminal check **?@sec-open-terminal**), and use the `pip` command to do it:

```
Terminal$ pip install pyspark
```

If you try to run the above command (inside a terminal of any OS), and a message like `pip: command not found` appears, this means that you do not have the `pip` tool installed on your machine. Hence, if you face this kind of message, you need to install `pip` before you even install `pyspark`.

The `pip` tool is automatically installed with Python on Windows. So, if you face this message (`pip: command not found`), then, is very likely that you do not have Python correctly installed on your machine. Or maybe, Python is not installed at all in any shape or size in your system. So, you should comeback to previous section, and re install it.

In UNIX-like systems, installing `pip` is very easy, because you can use the built-in package manager to do this for you. In Debian like distros (e.g. Ubuntu), you use the `apt` tool, and,

[4] https://www.python.org/downloads/

in Arch-Linux like distros (e.g. Arch Linux, Manjaro) you would use the `pacman` tool. Both possibilities are exposed below:

```
# If you are in a Debian like distro of Linux
# and need to install `pip`, use this command:
Terminal$ apt install python3-pip
# If you are in a Arch-Linux like distro of Linux
# and need to install `pip`, use this command:
Terminal$ pacman -S python-pip
```

But, if you are running in MacOs, is best to use the `get-pip.py` method to do install `pip`[5]. To use this method, try to run the command below on the terminal. After you installed pip, run the same `pip` command we showed earlier to install `pyspark`.

```
# To install `pip` on a MacOs:
Terminal$ curl https://bootstrap.pypa.io/get-pip.py | python3
```

After you correctly installed `pip`, try to execute the first command again, to install the `pyspark` package on your machine.

Install `Spark`

[5]See the following discussion: https://stackoverflow.com/questions/17271319/how-do-i-install-pip-on-macos-or-os-x

1 Key concepts of python

1.1 Introduction

If you have experience with python, and understands how objects and classes works, you might want to skip this entire chapter. But, if you are new to the language and do not have much experience with it, you might want to stick a little bit, and learn a few key concepts that will help you to understand how the `pyspark` package is organized, and how to work with it.

1.2 Scripts

Python programs are written in plain text files that are saved with the `.py` extension. After you save these files, they are usually called "scripts". So a script is just a text file that contains all the commands that make your python program.

There are many IDEs or programs that help you to write, manage, run and organize this kind of files (like Microsoft Visual Studio Code[1], PyCharm[2], Anaconda[3] and RStudio[4]). But, if you do not have any of them installed, you can just create a new plain text file from the built-in Notepad program of your OS (operational system), and, save it with the `.py` extension.

1.3 How to run a python program

As you learn to write your Spark applications with `pyspark`, at some point, you will want to actually execute this `pyspark` program, to see its result. To do so, you need to execute it as a python program. There are many ways to run a python program, but I will show you the more "standard" way. That is to use the `python` command inside the terminal of your OS (you need to have python already installed).

As an example, lets create a simple "Hello world" program. First, open a new text file then save it somewhere in your machine (with the name `hello.py`). Remember to save the file with the `.py` extension. Then copy and paste the following command into this file:

[1] https://code.visualstudio.com/
[2] https://www.jetbrains.com/pycharm/
[3] https://www.anaconda.com/products/distribution
[4] https://www.rstudio.com/

```
print("Hello World!")
```

It will be much easier to run this script, if you open the terminal inside the folder where you save the `hello.py` file. If you do not know how to do this, look at section **?@sec-open-terminal**. After you opened the terminal inside the folder, just run the `python hello.py` command. As a result, python will execute `hello.py`, and, the text `Hello World!` should be printed to the terminal:

```
Terminal$ python hello.py
```

```
Hello World!
```

But, if for some reason you could not open the terminal inside the folder, just open a terminal (in any way you can), then, use the `cd` command (stands for "change directory") with the path to the folder where you saved `hello.py`. This way, your terminal will be rooted in this folder.

For example, if I saved `hello.py` inside my Documents folder, the path to this folder in Windows would be something like this: `"C:\Users\pedro\Documents"`. On the other hand, this path on Linux would be something like `"/usr/pedro/Documents"`. So the command to change to this directory would be:

```
# On Windows:
Terminal$ cd "C:\Users\pedro\Documents"
# On Linux:
Terminal$ cd "/usr/pedro/Documents"
```

After this `cd` command, you can run the `python hello.py` command in the terminal, and get the exact same result of the previous example.

There you have it! So every time you need to run your python program (or your `pyspark` program), just open a terminal and run the command `python <complete path to your script>`. If the terminal is rooted on the folder where you saved your script, you can just use the `python <name of the script>` command.

1.4 Objects

Although python is a general-purpose language, most of its features are focused on object-oriented programming. Meaning that, python is a programming language focused on creating, managing and modifying objects and classes of objects.

So, when you work with python, you are basically applying many operations and functions over a set of objects. In essence, an object in python, is a name that refers to a set of data. This data can be anything that you computer can store (or represent).

Having that in mind, an object is just a name, and this name is a reference, or a key to access some data. To define an object in python, you must use the assignment operator, which is the equal sign (`=`). In the example below, we are defining, or, creating an object called `x`, and it stores the value 10. Therefore, with the name `x` we can access this value of 10.

```
x = 10
print(x)
```

```
10
```

When we store a value inside an object, we can easily reuse this value in multiple operations or expressions:

```
# Multiply by 2
print(x * 2)
```

```
20
```

```
# Divide by 3
print(x / 3)
```

```
3.3333333333333335
```

```
# Print its class
print(type(x))
```

```
<class 'int'>
```

Remember, an object can store any type of value, or any type of data. For example, it can store a single string, like the object `salutation` below:

```
salutation = "Hello! My name is Pedro"
```

Or, a list of multiple strings:

```
names = [
  "Anne", "Vanse", "Elliot",
  "Carlyle", "Ed", "Memphis"
]

print(names)
```

```
['Anne', 'Vanse', 'Elliot', 'Carlyle', 'Ed', 'Memphis']
```

Or a dict containing the description of a product:

```
product = {
  'name': 'Coca Cola',
  'volume': '2 litters',
  'price': 2.52,
  'group': 'non-alcoholic drinks',
  'department': 'drinks'
}

print(product)
```

```
{'name': 'Coca Cola', 'volume': '2 litters', 'price': 2.52, 'group': 'non-alcoholic drinks',
```

And many other things...

1.5 Expressions

Python programs are organized in blocks of expressions (or statements). A python expression is a statement that describes an operation to be performed by the program. For example, the expression below describes the sum between 3 and 5.

```
3 + 5
```

```
8
```

The expression above is composed of numbers (like 3 and 5) and a operator, more specifically, the sum operator (`+`). But any python expression can include a multitude of different items. It can be composed of functions (like `print()`, `map()` and `str()`), constant strings (like `"Hello World!"`), logical operators (like `!=`, `<`, `>` and `==`), arithmetic operators (like `*`, `/`, `**`, `%`, `-` and `+`), structures (like lists, arrays and dicts) and many other types of commands.

Below we have a more complex example, that contains the `def` keyword (which starts a function definition; in the example below, this new function being defined is `double()`), many built-in functions (`list()`, `map()` and `print()`), a arithmetic operator (`*`), numbers and a list (initiated by the pair of brackets - `[]`).

```
def double(x):
  return x * 2

print(list(map(double, [4, 2, 6, 1])))
```

```
[8, 4, 12, 2]
```

Python expressions are evaluated in a sequential manner (from top to bottom of your python file). In other words, python runs the first expression in the top of your file, them, goes to the second expression, and runs it, them goes to the third expression, and runs it, and goes on and on in that way, until it hits the end of the file. So, in the example above, python executes the function definition (initiated at `def double(x):`), before it executes the `print()` statement, because the print statement is below the function definition.

This order of evaluation is commonly referred as "control flow" in many programming languages. Sometimes, this order can be a fundamental part of the python program. Meaning that, sometimes, if we change the order of the expressions in the program, we can produce unexpected results (like an error), or change the results produced by the program.

As an example, the program below prints the result 4, because the print statement is executed before the expression `x = 40`.

```
x = 1

print(x * 4)

x = 40
```

```
4
```

But, if we execute the expression `x = 40` before the print statement, we then change the result produced by the program.

```
x = 1
x = 40

print(x * 4)
```

```
160
```

If we go a little further, and, put the print statement as the first expression of the program, we then get a name error. This error warns us that, the object named `x` is not defined (i.e. it does not exist).

```
print(x * 4)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

```
x = 1
x = 40
```

This error occurs, because inside the print statement, we call the name `x`. But, this is the first expression of the program, and at this point of the program, we did not defined a object called `x`. We make this definition, after the print statement, with `x = 1` and `x = 40`. In other words, at this point, python do not know any object called `x`.

1.6 Packages

A python package (or a python "library") is basically a set of functions and classes that provides important functionality to solve a specific problem. And `pyspark` is one of these many python packages available.

Python packages are usually published (that is, made available to the public) through the PyPI archive[5]. If a python package is published in PyPI, then, you can easily install it through the `pip` tool, that we just used in **?@sec-install-software**.

[5] https://pypi.org/

To use a python package, you always need to: 1) have this package installed on your machine; 2) import this package in your python script. If a package is not installed in your machine, you will face a `ModuleNotFoundError` as you try to use it, like in the example below. So, if your program produce such an error, is very likely that you are trying to use a package that is not currently installed on your machine. To install it, you may use the `pip install <name of the package>` command on the terminal of your OS.

```
import a_package
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'a_package'
```

But, if this package is already installed in your machine, then, you can just import it to your script. To do this, you just include an `import` statement at the start of your python file. For example, if I want to use the `DataFrame` function from the `pandas` package:

```
import pandas

df = pandas.DataFrame([
  (1, 3214), (2, 4510),
  (1, 9082), (4, 7822)
])

print(df)
```

```
   0     1
0  1  3214
1  2  4510
2  1  9082
3  4  7822
```

Therefore, with `import pandas` I can access any of the functions available in the `pandas` package, by using the dot operator after the name of the package (`pandas.<name of the function>`). However, it can become very annoying to write `pandas.` every time you want to access a function from `pandas`, specially if you use it constantly in your code.

To make life a little easier, python offers some alternative ways to define this `import` statement. First, you can give an alias to this package that is shorter/easier to write. As an example, nowadays, is virtually a industry standard to import the `pandas` package as `pd`. To do this,

you use the `as` keyword in your `import` statement. This way, you can access the `pandas` functionality with `pd.<name of the function>`:

```
import pandas as pd

df = pd.DataFrame([
  (1, 3214), (2, 4510),
  (1, 9082), (4, 7822)
])

print(df)
```

```
   0     1
0  1  3214
1  2  4510
2  1  9082
3  4  7822
```

In contrast, if you want to make your life even easier and produce a more "clean" code, you can import (from the package) just the functions that you need to use. In this method, you can eliminate the dot operator, and refer directly to the function by its name. To use this method, you include the `from` keyword in your import statement, like this:

```
from pandas import DataFrame

df = DataFrame([
  (1, 3214), (2, 4510),
  (1, 9082), (4, 7822)
])

print(df)
```

```
   0     1
0  1  3214
1  2  4510
2  1  9082
3  4  7822
```

Some packages may be very big, and includes many different functions and classes. As the size of the package becomes bigger and bigger, developers tend to divide this package in may

"modules". In other words, the many functions and classes that are available in a python package, are usually organized in "modules".

As an example, the famous `flask` package is a fairly big package, that contains many functionalities. Because of it, the package is organized in a number of modules, such as `cli`, `globals`, `views` and `sessions`. To access the functions available in each one of these modules, you use the dot operator.

1.7 Methods versus Functions

Beginners tend mix these two types of functions in python, but they are not the same. So lets describe the differences between the two.

Standard python functions, are **functions that we apply over an object**. A classical example, is the `print()` function. You can see in the example below, that we are applying `print()` over the `result` object.

```
result = 10 + 54
print(result)
```

```
64
```

Other examples of a standard python function would be `map()` and `list()`. See in the example below, that we apply the `map()` function over a set of objects:

```
words = ['apple', 'star', 'abc']
lengths = map(len, words)
list(lengths)
```

```
[5, 4, 3]
```

In contrast, a python method is a function registered inside a python class. In other words, this function **belongs to the class itself**, and cannot be used outside of it. To use a method, you need to have an instance of the class where it is registered.

For example, the `startswith()` method belongs to the `str` class (this class is used to represent strings in python). So to use this method, we need to have an instance ot this class saved in a object that we can access. Note in the example below, that we access the `startswith()` method through the `name` object. This means that, `startswith()` is a function. But, we cannot use it without an object of class `str`, like `name`.

```
name = "Pedro"
name.startswith("P")
```

```
True
```

So, if we have a class called `people`, and, this class has a method called `location()`, we can use this `location()` method by using the dot operator (`.`) with the name of an object of class `people`. If an object called `x` is an instance of `people` class, then, we can do `x.location()`.

But if this object `x` is of a different class, like `int`, then we can no longer use the `location()` method, because this method does not belong to the `int` class. For example, if your object is from class `A`, and, you try to use a method of class `B`, you will get an `AttributeError`.

In the example exposed below, I have an object called `number` of class `int`, and, I try to use the method `startswith()` from `str` class with this object:

```
number = 2
# You can see below that, the `x` object have class `int`
type(number)
# Trying to use a method from `str` class
number.startswith("P")
```

```
AttributeError: 'int' object has no attribute 'startswith'
```

1.8 Identifying classes and their methods

Over the next chapters, you will realize that the `pyspark` API does not have many standard python functions. So most of its functionality resides in class methods. As a result, the capability of understanding the objects that you have in your python program, and, identifying its classes and methods will be crucial while you are developing and debugging your Spark applications.

Every existing object in python represents an instance of a class. In other words, every object in python is associated to a given class. You can always identify the class of an object, by applying the `type()` function over this object. In the example below, we can see that, the `name` object is an instance of the `str` class.

```
name = "Pedro"
type(name)
```

`str`

If you do not know all the methods that a class have, you can always apply the `dir()` function over this class to get a list of all available methods. For example, lets suppose you wanted to see all methods from the `str` class. To do so, you would do this:

```
dir(str)
```

```
['__add__', '__class__', '__contains__', '__delattr__', '__dir__',
 '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
 '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__',
 '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__',
 '__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
 '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__',
 '__str__', '__subclasshook__', 'capitalize', 'casefold', 'center',
 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format',
 'format_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal',
 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable',
 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip',
 'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust',
 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith',
 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

2 Introducing Apache Spark

2.1 Introduction

In essence, `pyspark` is an API to Apache Spark (or simply Spark). In other words, with `pyspark` we can build Spark applications using the python language. So, by learning a little more about Spark, you will understand a lot more about `pyspark`.

2.2 What is Spark?

Spark is a multi-language engine for large-scale data processing that supports both single-node machines and clusters of machines (*Apache Spark Official Documentation* 2022). Nowadays, Spark became the de facto standard for structure and manage big data applications.

It has a number of features that its predecessors did not have, like the capacity for in-memory processing and stream processing (Karau et al. 2015). But, the most important feature of all, is that Spark is an **unified platform** for big data processing (Chambers and Zaharia 2018). This means that Spark comes with multiple built-in libraries and tools that deals with different aspects of the work with big data. It has a built-in SQL engine[1] for performing large-scale data processing; a complete library for scalable machine learning (`MLib`[2]); a stream processing engine[3] for streaming analytics; and much more;

In general, big companies have many different data necessities, and as a result, the engineers and analysts may have to combine and integrate many tools and techniques together, so they can build many different data pipelines to fulfill these necessities. But this approach can create a very serious dependency problem, which imposes a great barrier to support this workflow. This is one of the big reasons why Spark got so successful. It eliminates big part of this problem, by already including almost everything that you might need to use.

> Spark is designed to cover a wide range of workloads that previously required separate distributed systems … By supporting these workloads in the same engine, Spark makes it easy and inexpensive to combine different processing types, which

[1] https://spark.apache.org/sql/
[2] https://spark.apache.org/docs/latest/ml-guide.html
[3] https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#overview

is often necessary in production data analysis pipelines. In addition, it reduces the management burden of maintaining separate tools (Karau et al. 2015).

2.3 Spark application

Your personal computer can do a lot of things, but, it cannot efficiently deal with huge amounts of data. For this situation, we need several machines working together, adding up their resources to deal with the volume or complexity of the data. Spark is the framework that coordinates the computations across this set of machines (Chambers and Zaharia 2018).

Because of this, a relevant part of Spark's structure is deeply connected to distributed computing models. You probably do not have a cluster of machines at home. So, while following the examples in this book, you will be running Spark on a single machine (i.e. single node mode). But lets just forget about this detail for a moment.

In every Spark application, you always have a single machine behaving as the driver node, and multiple machines behaving as the worker nodes. The driver node is responsible for managing the Spark application, i.e. asking for resources, distributing tasks to the workers, collecting and compiling the results, ... The worker nodes are responsible for executing the tasks that are assigned to them, and they need to send the results of these tasks back to the driver node.

Every Spark application is distributed into two different and independent processes: 1) a driver process; 2) and a set of executor processes (Chambers and Zaharia 2018). The driver process, or, the driver program, is where your application starts, and it is executed by the driver node. This driver program is responsible for: 1) maintaining information about your Spark Application; 2) responding to a user's program or input; 3) and analyzing, distributing, and scheduling work across the executors (Chambers and Zaharia 2018).

Every time a Spark application starts, the driver process has to communicate with the cluster manager, to acquire workers to perform the necessary tasks. In other words, the cluster manager decides if Spark can use some of the resources (i.e. some of the machines) of the cluster. If the cluster manager allow Spark to use the nodes it needs, the driver program will break the application into many small tasks, and will assign these tasks to the worker nodes.

The executor processes, are the processes that take place within each one of the worker nodes. Each executor process is composed of a set of tasks, and the worker node is responsible for performing and executing these tasks that were assigned to him, by the driver program. After executing these tasks, the worker node will send the results back to the driver node (or the driver program). If they need, the worker nodes can communicate with each other, while performing its tasks.

This structure is represented in Figure 2.1:

When you run Spark on a cluster of computers, you write the code of your Spark application (i.e. your `pyspark` code) on your (single) local computer, and then, submit this code to the

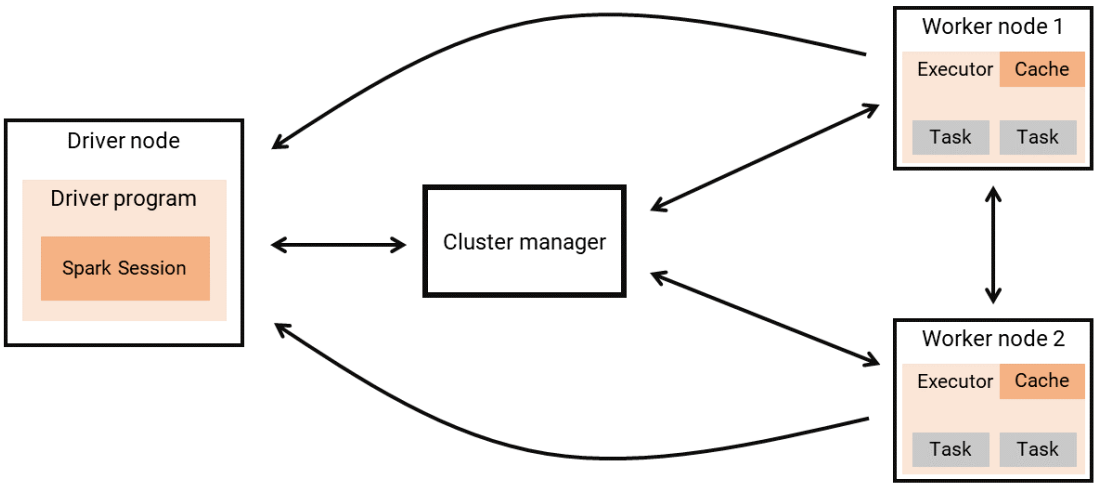

Figure 2.1: Spark application structure on a cluster of computers

driver node. After that, the driver node takes care of the rest, by starting your application, creating your Spark Session, asking for new worker nodes, sending the tasks to be performed, collecting and compiling the results and giving back these results to you.

However, when you run Spark on your (single) local computer, the process is very similar. But, instead of submitting your code to another computer (which is the driver node), you will submit to your own local computer. In other words, when Spark is running on single-node mode, your computer becomes the driver and the worker node at the same time.

2.4 Starting your Spark Session

Every Spark application starts with a Spark Session. Basically, the Spark Session is the entry point to your application. This means that, in every `pyspark` program that you write, **you should always start by defining your Spark Session**. We do this, by using the `getOrCreate()` method from `pyspark.sql.SparkSession.builder`.

Just store the result of this method in any python object. Is very common to name this object as `spark`, like in the example below. This way, you can access all the information and methods of Spark from this `spark` object.

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()
```

```
22/06/14 19:11:13 WARN Utils: Your hostname, pedro-IdeaPad-3-15ALC6 resolves to a loopback ad
22/06/14 19:11:13 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another address

Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).

22/06/14 19:11:14 WARN NativeCodeLoader: Unable to load native-hadoop library for your platfo
```

2.5 Running your first Spark application

To demonstrate what I `pyspark` program looks like, lets write and run a very simple example of Spark application. This Spark application will build a simple table of 1 column that contains 5 numbers, and then, it will return a list with the first two rows of this table as the result.

First, create a new blank text file in your computer, and save it somewhere with the name `spark-example.py`. Do not forget to put the `.py` extension in the name. This program we are writing together is a python program, and should be treated as such. With the `.py` extension in the name file, you are stating this fact quite clearly to your computer.

After you created and saved the python script (i.e. the text file with the `.py` extension), you can start writing your `pyspark` program. As we noted in the previous section, you should always start your `pyspark` program by defining your Spark Session, with this code:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()
```

After you defined your Spark Session, and saved it in an object called `spark`, you can now access all Spark's functionality through this `spark` object. To create the table we use the `range()` method, and to return the first two rows of the resulting table, we use the `take()` method:

```
table = spark.range(5)
table.take(2)
```

So, the entire program is just these two parts (or sections) of code. Just copy and paste this code to your python script, then save it.

```
# The entire program:
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

table = spark.range(5)
table.take(2)
```

Now that you have written your first Spark application with `pyspark`, you want to execute this application and see its results. To do so, you need send this script to the python interpreter, and to do this you need to: 1) copy the path to your python script; and, 2) open a terminal.

3 Introducing Spark DataFrames

3.1 Introduction

In this chapter, you will understand how Spark represents and manages tables (or tabular data). Different programming languages and frameworks use different names to describe a table. But, in Apache Spark, tables are referred as Spark DataFrames.

In `pyspark`, these DataFrames are stored inside python objects of class `pyspark.sql.dataframe.DataFrame`, and all the methods present in this class, are commonly referred as the DataFrame API of Spark. This is the most important API of Spark. Much of your Spark applications will heavily use this API to compose your data transformations and data flows (Chambers and Zaharia 2018).

3.2 Spark DataFrames versus Spark Datasets

Spark have two notions of structured data: DataFrames and Datasets. In summary, a Spark Dataset, is a distributed collection of data (*Apache Spark Official Documentation* 2022). In contrast, a Spark DataFrame is a Spark Dataset organized into named columns (*Apache Spark Official Documentation* 2022).

This means that, Spark DataFrames are very similar to tables as we know in relational databases - RDBMS. So in a Spark DataFrame, each column has a name, and they all have the same number of rows. Furthermore, all the rows inside a column must store the same type of data, but each column can store a different type of data.

In the other hand, Spark Datasets are considered a collection of any type of data. So a Dataset might be a collection of unstructured data as well, like log files, JSON and XML trees, etc. Spark Datasets can be created and transformed trough the Dataset API of Spark. But this API is available only in Scala and Java API's of Spark. For this reason, we do not act directly on Datasets with `pyspark`, only DataFrames. That's ok, because for the most part of applications, we do want to use DataFrames, and not Datasets, to represent our data.

However, what makes a Spark DataFrame different from other dataframes? Like the `pandas` DataFrame? Or the R native `data.frame` structure? Is the **distributed** aspect of it. Spark DataFrames are based on Spark Datasets, and these Datasets are collections of data that are

distributed across the cluster. As an example, lets suppose you have the following table stored as a Spark DataFrame:

ID	Name	Value
1	Anne	502
2	Carls	432
3	Stoll	444
4	Percy	963
5	Martha	123
6	Sigrid	621

If you are running Spark in a 4 nodes cluster (one is the driver node, and the other three are worker nodes). Each worker node of the cluster will store a section of this data. So you, as the programmer, will see, manage and transform this table as if it was a single and unified table. But behind the hoods, Spark will split this data and store it as many fragments across the Spark cluster. Figure 3.1 presents this notion in a visual manner.

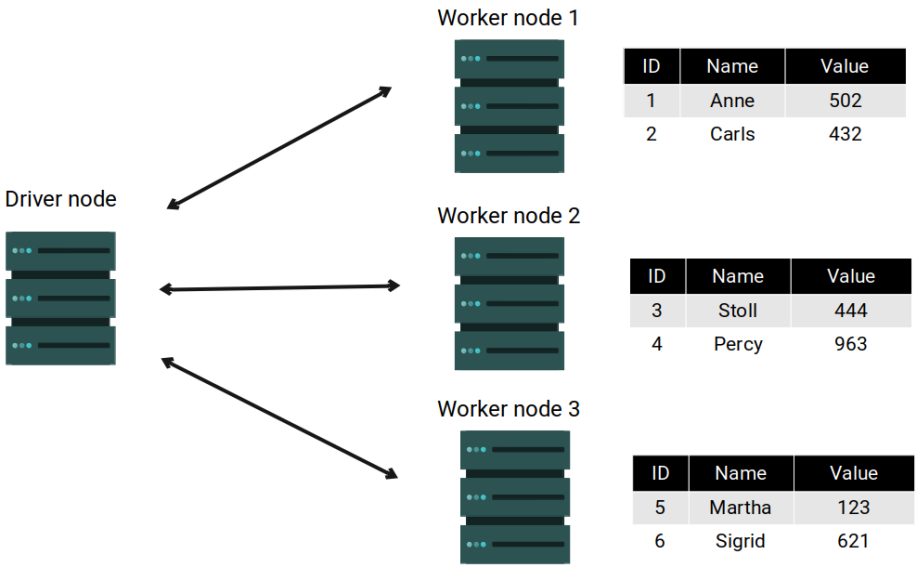

Figure 3.1: A Spark DataFrame is distributed across the cluster

3.3 Building a Spark DataFrame

There are some different methods to create a Spark DataFrame. For example, because a DataFrame is basically a Dataset of rows, we can build a DataFrame from a collection of `Row`'s, through the `createDataFrame()` method from your Spark Session:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()
from datetime import date
from pyspark.sql import Row

data = [
  Row(id = 1, value = 28.3, date = date(2021,1,1)),
  Row(id = 2, value = 15.8, date = date(2021,1,1)),
  Row(id = 3, value = 20.1, date = date(2021,1,2)),
  Row(id = 4, value = 12.6, date = date(2021,1,3))
]

df = spark.createDataFrame(data)
```

```
22/06/14 19:11:21 WARN Utils: Your hostname, pedro-IdeaPad-3-15ALC6 resolves to a loopback ac
22/06/14 19:11:21 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another address

Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).

22/06/14 19:11:22 WARN NativeCodeLoader: Unable to load native-hadoop library for your platfo
```

Remember that a Spark DataFrame in python is a object of class `pyspark.sql.dataframe.DataFrame` as you can see below. If you try to see what is inside of this kind of object, you will get a small description of the columns present in the DataFrame as a result:

```
type(df)
```

```
<class 'pyspark.sql.dataframe.DataFrame'>
```

```
df
```

```
DataFrame[id: bigint, value: double, date: date]
```

So, in the above example, we use the `Row()` constructor (from `pyspark.sql` module) to build 4 rows. The `createDataFrame()` method, stack these 4 rows together to form our new DataFrame `df`. The result is a Spark DataFrame with 4 rows and 3 columns (`id`, `value` and `date`).

But you can use different methods to create the same Spark DataFrame. As another example, with the code below, we are creating a DataFrame called `students`. To do this, we create two python lists (`data` and `columns`), then, deliver these lists to `createDataFrame()` method. Each element of `data` is a python `tuple` that represents a row in the `students` DataFrame. And each element of `columns` represent the name of a column present in this new DataFrame.

```
data = [
  (12114, 'Anne', 21, 1.56, 8, 9, 10, 9, 'Economics', 'SC'),
  (13007, 'Adrian', 23, 1.82, 6, 6, 8, 7, 'Economics', 'SC'),
  (10045, 'George', 29, 1.77, 10, 9, 10, 7, 'Law', 'SC'),
  (12459, 'Adeline', 26, 1.61, 8, 6, 7, 7, 'Law', 'SC'),
  (10190, 'Mayla', 22, 1.67, 7, 7, 7, 9, 'Design', 'AR'),
  (11552, 'Daniel', 24, 1.75, 9, 9, 10, 9, 'Design', 'AR')
]

columns = [
  'StudentID', 'Name', 'Age', 'Height', 'Score1',
  'Score2', 'Score3', 'Score4', 'Course', 'Department'
]

students = spark.createDataFrame(data, columns)
students
```

```
DataFrame[StudentID: bigint, Name: string, Age: bigint, Height: double, Score1: bigint, Score
```

If you need to, you can easily collect a python list with the column names present in your DataFrame, in the same way you would do in a `pandas` DataFrame. That is, by using the `columns` method of your DataFrame, like this:

```
students.columns
```

```
['StudentID',
 'Name',
 'Age',
```

```
 'Height',
 'Score1',
 'Score2',
 'Score3',
 'Score4',
 'Course',
 'Department']
```

3.4 Viewing a Spark DataFrame

A key aspect of Spark is its laziness. In other words, for most operations, Spark will only check if your code is correct and if it makes sense. Spark will not actually run or execute the operations you are describing in your code, unless you explicit ask for it with a trigger operation, that is called an "action" (this kind of operation is described in Section 4.3).

You can notice this laziness in the above output. Because when we call for an object that stores a Spark DataFrame (like `df` and `students`), Spark will only calculate and print a summary of the structure of your Spark DataFrame, and not the DataFrame itself.

So how can we actually see our DataFrame? How can we visualize the rows and values that are stored inside of it? We use the `show()` method. With this method, Spark will print the table as pure text, as you can see in the example below:

```
students.show()
```

```
[Stage 0:>                                                          (0 + 1) / 1]
```

```
+---------+-------+---+------+------+------+------+------+---------+----------+
|StudentID|   Name|Age|Height|Score1|Score2|Score3|Score4|   Course|Department|
+---------+-------+---+------+------+------+------+------+---------+----------+
|    12114|   Anne| 21|  1.56|     8|     9|    10|     9|Economics|        SC|
|    13007| Adrian| 23|  1.82|     6|     6|     8|     7|Economics|        SC|
|    10045| George| 29|  1.77|    10|     9|    10|     7|      Law|        SC|
|    12459|Adeline| 26|  1.61|     8|     6|     7|     7|      Law|        SC|
|    10190|  Mayla| 22|  1.67|     7|     7|     7|     9|   Design|        AR|
|    11552| Daniel| 24|  1.75|     9|     9|    10|     9|   Design|        AR|
+---------+-------+---+------+------+------+------+------+---------+----------+
```

By default, this method shows only the top rows of your DataFrame, but you can specify how much rows exactly you want to see, by using `show(n)`, where `n` is the number of rows. For example, I can visualize only the first 2 rows of `df` like this:

```
df.show(2)
```

```
+---+-----+----------+
| id|value|      date|
+---+-----+----------+
|  1| 28.3|2021-01-01|
|  2| 15.8|2021-01-01|
+---+-----+----------+
only showing top 2 rows
```

3.5 Spark Data Types

Each column of your Spark DataFrame is associated with a specific data type. Spark supports a number of different data types. You can see the full list at the official documentation page[1]. For now, we will focus on the most used data types, which are listed below:

- `IntegerType`: Represents 4-byte signed integer numbers. The range of numbers is from -2147483648 to 2147483647.
- `LongType`: Represents 8-byte signed integer numbers. The range of numbers is from -9223372036854775808 to 9223372036854775807.
- `FloatType`: Represents 4-byte single-precision floating point numbers.
- `DoubleType`: Represents 8-byte double-precision floating point numbers.
- `StringType`: Represents character string values.
- `BooleanType`: Represents boolean values (true or false).
- `TimestampType`: Represents datetime values, i.e. values that contains fields year, month, day, hour, minute, and second, with the session local time-zone. The timestamp value represents an absolute point in time.
- `DateType`: Represents date values, i.e. values that contains fields year, month and day, without a time-zone.

Besides these more "standard" data types, Spark supports two other complex types, which are `ArrayType` and `MapType`:

- `ArrayType(elementType, containsNull)`: Represents a sequence of elements with the type of `elementType`. `containsNull` is used to indicate if elements in a `ArrayType` value can have `null` values.
- `MapType(keyType, valueType, valueContainsNull)`: Represents a set of key-value pairs. The data type of keys is described by `keyType` and the data type of values is

[1] spark-data-types1

described by `valueType`. For a `MapType` value, keys are not allowed to have `null` values. `valueContainsNull` is used to indicate if values of a `MapType` value can have `null` values.

Each one of these Spark data types have a corresponding python class in `pyspark`, which are stored in the `pyspark.sql.types` module. As a result, to access, lets say, type `StryngType`, we can do this:

```
from pyspark.sql.types import StringType
s = StringType()
print(s)
```

```
StringType
```

3.6 The DataFrame Schema

The schema of a Spark DataFrame is the combination of column names and the data types associated with each of these columns. Schemas can be set explicitly by you (that is, you can tell Spark how the schema of your DataFrame should look like), or, can be defined automatically by Spark while reading and creating data. You can get a succinct description of a DataFrame schema, by looking inside the object where this DataFrame is stored.

For example, lets look again to the `df` DataFrame. In the result below, we can see that `df` has three columns (`id`, `value` and `date`). By the description `id: bigint`, we know that `id` is a column of type `bigint`, which translates to the `LongType()` of Spark (we will talk shortly about these data types). Furthermore, by the descriptions `value: double` and `date: date`, we know too that the columns `value` and `date` are of type `double` and `date`, respectively.

```
df
```

```
DataFrame[id: bigint, value: double, date: date]
```

3.6.1 Accessing the DataFrame schema

So, by calling the object of your DataFrame (i.e. an object of class `pyspark.sql.dataframe.DataFrame`) in the console, you can see a small description of the schema of this DataFrame. But, how can you access the description of this schema programmatically? You do this, by using the `schema` method of your DataFrame, like in the example below:

```
df.schema
```

```
StructType(List(StructField(id,LongType,true),StructField(value,DoubleType,true),StructField
```

The result of the `schema` method, is a `StructType()` object, that contains some information about each column of your DataFrame. More specifically, a `StructType()` object is filled with multiple `StructField()` objects. Each `StructField()` object store the name and the type of a column, and a boolean value (`True` or `False`) that indicates if this column can contain any null value inside of it.

You can use a `for` loop to iterate through this `StructType()` and get the information of each column separately.

```
schema = df.schema
for column in schema:
  print(column)
```

```
StructField(id,LongType,true)
StructField(value,DoubleType,true)
StructField(date,DateType,true)
```

You can access just the data type of each column by using the `dataType` method of each `StructField()` object.

```
for column in schema:
  datatype = column.dataType
  print(datatype)
```

```
LongType
DoubleType
DateType
```

And you can do the same for column names and the boolean value (that indicates if the column can contain "null" values), by using the `name` and `nullable` methods, respectively.

```
# Accessing the name of each column
for column in schema:
  print(column.name)
```

```
id
value
date
```

```
# Accessing the boolean value that indicates
# if the column can contain null values
for column in schema:
  print(column.nullable)
```

```
True
True
True
```

3.6.2 Building a DataFrame schema

When Spark creates a new DataFrame, it will automatically guess which schema is appropriate for that DataFrame. In other words, Spark will try to guess which are the appropriate data types for each column. But, this is just a guess, and, sometimes, Spark go way off.

Because of that, in some cases, you have to tell Spark how exactly you want this DataFrame schema to be like. To do that, you need to build the DataFrame schema by yourself, with `StructType()` and `StructField()` functions, alongside with the Spark data types (i.e. `StringType()`, `DoubleType()`, `IntegerType()`, ...). Remember, all of these python classes come from the `pyspark.sql.types` module.

In the example below, the `schema` object represents the schema of the `registers` DataFrame. This DataFrame have three columns (`ID`, `Date`, `Name`) of types `IntegerType`, `DateType` and `StringType`, respectively. See below that I deliver this `schema` object that I built to `spark.createDataFrame()`.

```
from pyspark.sql.types import StructType, StructField
from pyspark.sql.types import DateType, StringType, IntegerType
from datetime import date

data = [
  (1, date(2022, 1, 1), 'Anne'),
  (2, date(2022, 1, 3), 'Layla'),
  (3, date(2022, 1, 15), 'Wick'),
  (4, date(2022, 1, 11), 'Paul')
]

schema = StructType([
  StructField('ID', IntegerType(), True),
  StructField('Date', DateType(), True),
  StructField('Name', StringType(), True)
```

```
])

registers = spark.createDataFrame(data, schema = schema)
```

So to build a DataFrame schema, you build the equivalent `StructType()` object that represents the schema you want.

3.6.3 Checking your DataFrame schema

In some cases, you need to include in your `pyspark` program, some checks that certifies that your Spark DataFrame have the expected schema. In other words, you want to take actions in case your DataFrame have a different schema that might cause a problem in your program.

To check if a specific column of your DataFrame is associated with the correct data type, you have to use the DataFrame schema to check if the respective column is an "instance" of the python class that represents that data type. Lets get back to the `df` DataFrame.

As an example, suppose you wanted to check if the `id` column is of type `IntegerType`. To do this check, we use the python built-in function `isinstance()` with the python class that represents the Spark `IntegerType` data type. As you can see below, the `id` column is not of type `IntegerType`.

```
from pyspark.sql.types import IntegerType
schema = df.schema
id_column = schema[0]
isinstance(id_column.dataType, IntegerType)
```

```
False
```

The `id` column is actually from the "big integer" type, or, the `LongType` type (which are 8-byte signed integer).

```
from pyspark.sql.types import LongType
isinstance(id_column.dataType, LongType)
```

```
True
```

3.7 The `Column` class

As we described in the introduction to this chapter, you will massively use the methods from the `pyspark.sql.dataframe.DataFrame` class in your Spark applications to manage, modify and calculate your Spark DataFrames. But there is one more python class that provides some very useful methods that you will regularly use, which is the `Column` class, or more specifically, the `pyspark.sql.column.Column` class.

The `Column` class is used to represent a column in a Spark DataFrame. This means that, each column of a Spark DataFrame is a object of class `Column`. We can confirm this statement, by looking at the class of any column from the `df` DataFrame.

```
type(df.id)
```

```
pyspark.sql.column.Column
```

You can refer or create a column, by using the `col()` function from `pyspark.sql.functions` module. In other words, the result of this `col()` function is always a object of class `Column`. The code below creates a column called `ID`.

```
from pyspark.sql.functions import col
a_column = col('ID')
print(a_column)
```

```
Column<'ID'>
```

You will see some of the methods from this `Column` class across the next chapters, like `desc()`, `alias()` and `cast()`.

3.8 Partitions of a Spark DataFrame

As we exposed in Figure 3.1, a Spark DataFrame is broken into many small parts that are distributed across the cluster. Each one of these parts are considered a DataFrame *partition*.

4 Transforming your Spark DataFrame

4.1 Introduction

Virtually every data analysis or data pipeline will include some ETL (*Extract, Transform, Load*) process, and the T is an essential part of it. Because, you almost never have an input data, or a initial DataFrame that perfectly fits your needs.

That means, that you almost always have to transform the initial data that you have, to a specific format that you can use in your analysis. In this chapter, you will learn how to apply some of these basic transformations to your Spark DataFrame.

4.2 Defining transformations

Spark DataFrames are immutable, meaning that, they cannot be directly changed. But you can use an existing DataFrame to create a new one, based on a set of transformations. In other words, you define a new DataFrame as a transformed version of an older DataFrame.

Basically every `pyspark` program that you write will have such transformations. Spark support many types of transformations, however, in this chapter, we will focus on four basic transformations that you can apply to a DataFrame:

- Filtering rows;
- Sorting rows;
- Adding or deleting columns;
- Calculate aggregates;

Therefore, when you apply one of the above transformations to an existing DataFrame, you will get a new DataFrame as a result. You usually combine multiple transformations together to get your desired result. As a first example, lets get back to the `df` DataFrame:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()
from datetime import date
from pyspark.sql import Row
```

```
data = [
  Row(id = 1, value = 28.3, date = date(2021,1,1)),
  Row(id = 2, value = 15.8, date = date(2021,1,1)),
  Row(id = 3, value = 20.1, date = date(2021,1,2)),
  Row(id = 4, value = 12.6, date = date(2021,1,3))
]

df = spark.createDataFrame(data)
```

```
22/06/14 19:11:34 WARN Utils: Your hostname, pedro-IdeaPad-3-15ALC6 resolves to a loopback ad
22/06/14 19:11:34 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another address

Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).

22/06/14 19:11:34 WARN NativeCodeLoader: Unable to load native-hadoop library for your platfo
```

In the example below, to create a new DataFrame called `big_values`, we begin with the `df` DataFrame, then, we filter its rows where `value` is greater than 15, then, we select `date` and `value` columns, then, we sort the rows based on the `value` column. So, this set of sequential transformations (filter it, then, select it, then, order it, ...) defines what this new `big_values` DataFrame is.

```
from pyspark.sql.functions import col
# You define a chain of transformations to
# create a new DataFrame
big_values = df\
  .filter(col('value') > 15)\
  .select('date', 'value')\
  .orderBy('value')
```

Thus, to apply a transformation to an existing DataFrame, we use DataFrame methods such as `select()`, `filter()`, `withColumn()`, `orderBy()` and `agg()`. Remember, these are methods from the python class that defines Apache Spark DataFrame's (i.e. `pyspark.sql.dataframe.DataFrame`). This means that you can apply these transformations only to Spark DataFrames, and no other kind of python object.

Each one of these methods create a *lazily evaluated transformation*. Once again, we see the **lazy** aspect of Spark doing its work here. All these transformation methods are lazily evaluated, meaning that, Spark will only check if they make sense with the initial DataFrame that you

have. Spark will not actually perform these transformations on your initial DataFrame, not untill you trigger these transformations with an **action**.

4.3 Triggering calculations with actions

Therefore, Spark will avoid performing any heavy calculation until such calculation is really needed. But how or when Spark will face this decision? **When it encounters an action**. An action is the tool you have to trigger Spark to actually perform the transformations you have defined.

> An action instructs Spark to compute the result from a series of transformations. (Chambers and Zaharia 2018).

There are four kinds of actions in Spark:

- Showing an output in the console;
- Writing data to some file or data source;
- Collecting data from a Spark DataFrame to native objects in python (or Java, Scala, R, etc.);
- Counting the number of rows in a Spark DataFrame;

You already know the first type of action, because we used it before with the `show()` method. This `show()` method is an action by itself, because you are asking Spark to show some output to you. So we can make Spark to actually calculate the transformations that defines the `big_values` DataFrame, by asking Spark to show this DataFrame to us.

```
big_values.show()
```

```
[Stage 0:>                                                          (0 + 12) / 12]
```

```
+----------+-----+
|      date|value|
+----------+-----+
|2021-01-01| 15.8|
|2021-01-02| 20.1|
|2021-01-01| 28.3|
+----------+-----+
```

```
[Stage 0:====>                                                      (1 + 11) / 12]
```

Another very useful action is the `count()` method, that gives you the number of rows in a DataFrame. To be able to count the number of rows in a DataFrame, Spark needs to access this DataFrame in the first place. That is why this `count()` method behaves as an action. Spark will perform the transformations that defines `big_values` to access the actual rows of this DataFrame and count them.

```
big_values.count()
```

```
3
```

Furthermore, sometimes, you want to collect the data of a Spark DataFrame to use it inside python. In other words, sometimes you need to do some work that Spark cannot do by itself. To do so, you collect part of the data that is being generated by Spark, and store it inside a normal python object to use it in a standard python program.

That is what the `collect()` method do. It transfers all the data of your Spark DataFrame into a standard python list that you can easily access with python. More specifically, you get a python list full of `Row()` values:

```
data = big_values.collect()
print(data)
```

```
[Row(date=datetime.date(2021, 1, 1), value=15.8), Row(date=datetime.date(2021, 1, 2), value=1
```

The `take()` method is very similar to `collect()`. But you usually apply `take()` when you need to collect just a small section of your DataFrame (and not the entire thing), like the first `n` rows.

```
n = 1
first_row = big_values.take(n)
print(first_row)
```

```
[Row(date=datetime.date(2021, 1, 1), value=15.8)]
```

The last action would be the `write()` method, but we will explain this method latter, in the Chapter 6.

4.4 Understanding narrow and wide transformations

There are two kinds of transformations in Spark: narrow and wide transformations. Remember, a Spark DataFrame is divided into many small parts (called partitions), and, these parts are spread across the cluster. The basic difference between narrow and wide transformations, is if the transformation forces Spark to read data from multiple partitions to generate a single part of the result of that transformation.

More technically, narrow transformations are simply transformations where 1 input data (or 1 partition of the input DataFrame) contributes to only 1 partition of the output.

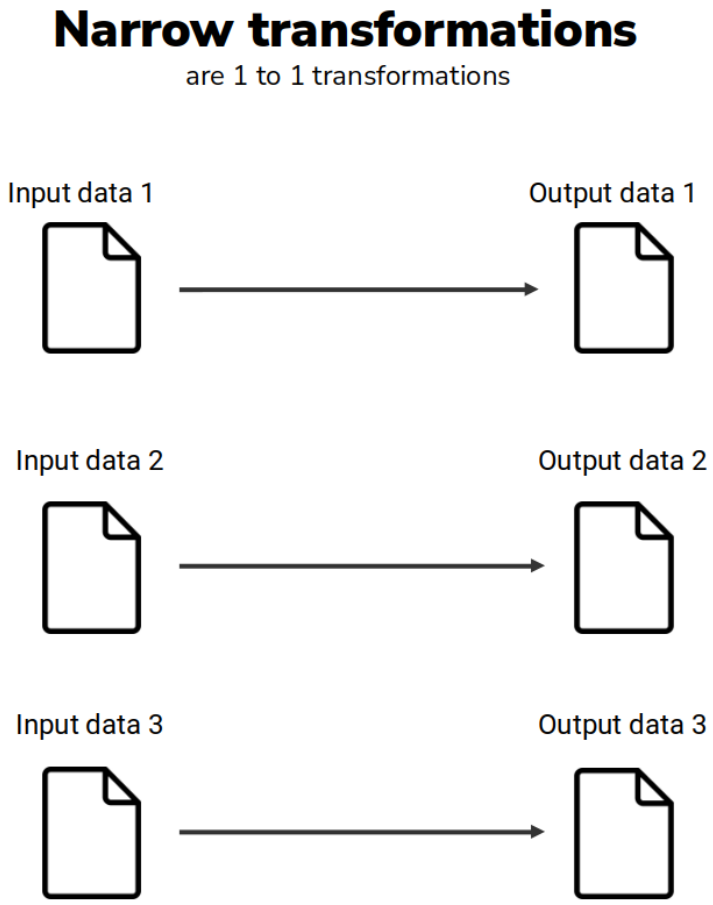

Figure 4.1: Presenting narrow transformations

In other words, each partition of your input DataFrame will be used (*separately*) to generate one individual part of the result of your transformation. As another perspective, you can

understand narrow transformations as those where Spark does not need to read the entire input DataFrame to generate a single and small piece of your result.

A classic example of narrow transformation is a filter. For example, suppose you have three students (Anne, Carls and Mike), and that each one has a bag full of blue, orange and red balls mixed. Now, suppose you asked them to collect all the red balls of these bags, and combined them in a single bag.

To do this task, Mike does not need to know what balls are inside of the bag of Carls or Anne. He just need to collect the red balls that are solely on his bag. At the end of the task, each student will have a part of the end result (that is, all the red balls that were in his own bag), and they just need to combine all these parts to get the total result.

The same thing applies to filters in Spark DataFrames. When you filter all the rows where the column `state` is equal to `"Alaska"`, Spark will filter all the rows in each partition separately, and then, will combine all the outputs to get the final result.

In contrast, wide transformations are the opposite of that. In wide transformations, Spark needs to use more than 1 partition of the input DataFrame to generate a small piece of the result.

When this kind of transformation happens, each worker node of the cluster needs to share his partition with the others. In other words, what happens is a partition shuffle. Each worker node sends his partition to the others, so they can have access to it, while performing their assigned tasks.

Partition shuffles are a very popular topic in Apache Spark, because they can be a serious source of inefficiency in your Spark application (Chambers and Zaharia 2018). In more details, when these shuffles happens, Spark needs to write data back to the hard disk of the computer, and this is not a very fast operation. It does not mean that wide transformations are bad or slow, just that the shuffles they are producing can be a problem.

A classic example of wide operation is a grouped aggregation. For example, lets suppose we had a DataFrame with the daily sales of multiple stores spread across the country, and, we wanted to calculate the total sales per city/region. To calculate the total sales of a specific city, like São Paulo, Spark would need to find all the rows that corresponds to this city, before adding the values, and these rows can be spread across multiple partitions of the cluster.

4.5 Filtering rows of your DataFrame

To filter specific rows of a DataFrame, `pyspark` offers two equivalent methods called `where()` and `filter()`. In other words, they both do the same thing, and work in the same way. You should give to these methods, a logical expression that translates what you want to filter.

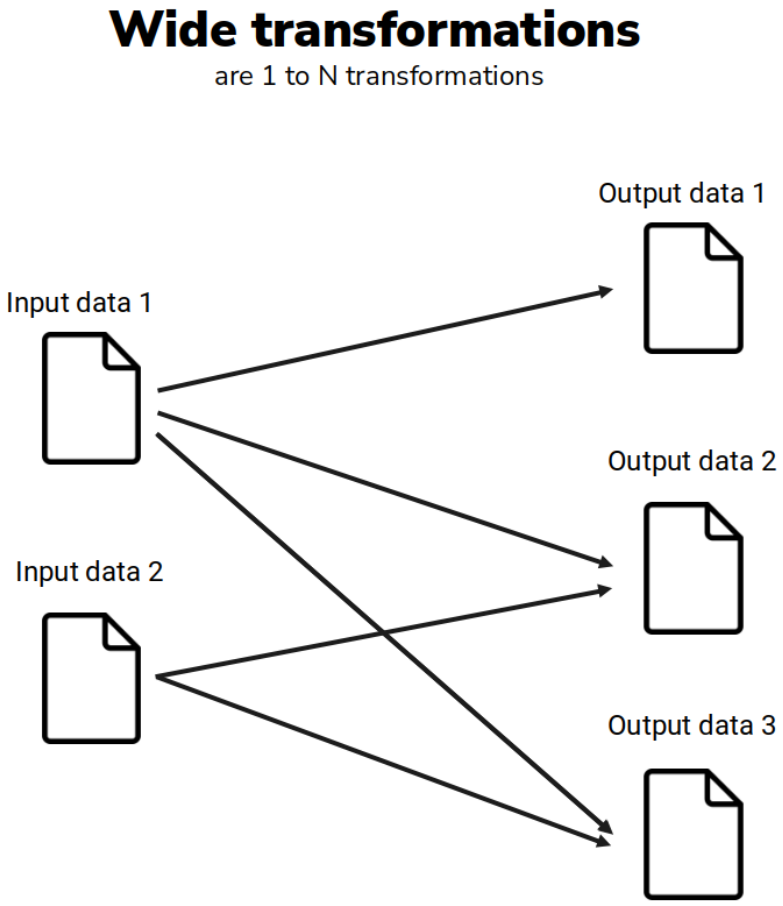

Figure 4.2: Presenting wide transformations

To demonstrate some of the examples in this and some of the next sections, we will use a different DataFrame, called `transf`. With the code below, you can import the data from `trans_reform.csv` to create this DataFrame in your Spark Session. Remember that, this CSV file is freely available to download at the repository of this book[1].

```
from pyspark.sql.types import StructType, StructField
from pyspark.sql.types import DoubleType, StringType, IntegerType, TimestampType
path = "../Data/transf_reform.csv"
schema = StructType([
  StructField('datetime', TimestampType(), False),
  StructField('user', StringType(), True),
  StructField('value', DoubleType(), True),
  StructField('transferid', IntegerType(), False),
  StructField('country', StringType(), True),
  StructField('description', StringType(), True)
])

transf = spark.read\
  .csv(path, schema = schema, sep = ";", header = True)
```

This `transf` DataFrame contains bank transfer records from a fictitious bank. Column `datetime` presents the date and time when the transfer occurred; `user` is the bank user that did the transfer; `value` presents the value that was transferred; `transferid` is an unique ID for the transfer; `country` show us the target country of the transfer; and `description` store some description of this transfer if needed.

```
transf.show(5)
```

```
+-------------------+--------+--------+----------+-------+-----------+
|           datetime|    user|   value|transferid|country|description|
+-------------------+--------+--------+----------+-------+-----------+
|2018-12-06 20:19:19| Eduardo|598.5984| 116241629|Germany|       null|
|2018-12-06 20:10:34|   Júlio|4610.955| 115586504|Germany|       null|
|2018-12-06 19:59:50|Nathália|4417.866| 115079280|Germany|       null|
|2018-12-06 19:54:13|   Júlio|2739.618| 114972398|Germany|       null|
|2018-12-06 19:41:27|     Ana|1408.261| 116262934|Germany|       null|
+-------------------+--------+--------+----------+-------+-----------+
only showing top 5 rows
```

[1] https://github.com/pedropark99/Introd-pyspark/tree/main/Data

As a first example, lets suppose you wanted to inspect all the rows where `value` is less than 1000. To do so, you can use the following code:

```
transf\
  .filter("value < 1000")\
  .show(5)
```

```
+-------------------+-----------+--------+----------+-------+-----------+
|           datetime|       user|   value|transferid|country|description|
+-------------------+-----------+--------+----------+-------+-----------+
|2018-12-06 20:19:19|    Eduardo|598.5984| 116241629|Germany|       null|
|2018-12-06 16:09:15|Júlio Cesar| 387.596| 114894102|Germany|       null|
|2018-12-06 12:47:25|        Ana|482.8322| 114387526|Germany|       null|
|2018-12-06 12:45:23|Júlio Cesar|909.7389| 114690249|Germany|       null|
|2018-12-06 11:50:11|    Armando|818.6205| 115148416|Germany|       null|
+-------------------+-----------+--------+----------+-------+-----------+
only showing top 5 rows
```

Writing simple SQL logical expression inside a string is the most easy and "clean" way to create a filter expression in `pyspark`. However, you could write the same exact expression in a more "pythonic" way, using the `col()` function from `pyspark.sql.functions`.

```
from pyspark.sql.functions import col

transf\
  .filter(col("value") < 1000)\
  .show(5)
```

```
+-------------------+-----------+--------+----------+-------+-----------+
|           datetime|       user|   value|transferid|country|description|
+-------------------+-----------+--------+----------+-------+-----------+
|2018-12-06 20:19:19|    Eduardo|598.5984| 116241629|Germany|       null|
|2018-12-06 16:09:15|Júlio Cesar| 387.596| 114894102|Germany|       null|
|2018-12-06 12:47:25|        Ana|482.8322| 114387526|Germany|       null|
|2018-12-06 12:45:23|Júlio Cesar|909.7389| 114690249|Germany|       null|
|2018-12-06 11:50:11|    Armando|818.6205| 115148416|Germany|       null|
+-------------------+-----------+--------+----------+-------+-----------+
only showing top 5 rows
```

You still have a more verbose alternative, that does not require the `col()` function. With this method, you refer to the specific column using the dot operator (`.`), like in the example below:

```
# This will give you the exact
# same result of the examples above
transf\
  .filter(transf.value < 1000)
```

4.5.1 Logical operators available

As we saw in the previous section, there are two ways to write logical expressions in `pyspark`: write SQL logical expressions inside a string; or, write python logical expressions using the `col()` function.

If you choose to write the SQL logical expressions in a string, you need to use the logical operators of SQL in your expression (not the logical operators of python). In the other hand, if you choose to write in the "python" way, then, you need to use the logical operators of python.

The logical operators of SQL are described in the table below:

Table 4.1: List of logical operators of SQL

Operator	Example of expression	Meaning of the expression
$<$	`x < y`	is `x` less than `y`?
$>$	`x > y`	is `x` greater than `y`?
$<=$	`x <= y`	is `x` less or equal than `y`?
$>=$	`x >= y`	is `x` greater or equal than `y`?
$==$	`x == y`	is `x` equal to `y`?
$!=$	`x != y`	is `x` not equal to `y`?
in	`x in y`	is `x` one of the values listed in `y`?
and	`x and y`	both logical expressions `x` and `y` are true?
or	`x or y`	at least one of logical expressions `x` and `y` are true?
not	`not x`	is the logical expression `x` not true?

The logical operators of python are described in the table below:

Table 4.2: List of logical operators of python

Operator	Example of expression	Meaning of the expression
<	`x < y`	is `x` less than `y`?
>	`x > y`	is `x` greater than `y`?
<=	`x <= y`	is `x` less or equal than `y`?
>=	`x >= y`	is `x` greater or equal than `y`?
==	`x == y`	is `x` equal to `y`?
!=	`x != y`	is `x` not equal to `y`?
&	`x & y`	both logical expressions `x` and `y` are true?
|	`x | y`	at least one of logical expressions `x` and `y` are true?
~	`~x`	is the logical expression `x` not true?

4.5.2 Connecting multiple logical expressions

Sometimes, you need to write more complex logical expressions to correctly describe the rows you are interested in. That is, when you combine multiple logical expressions together.

As an example, lets suppose you wanted all the rows in `transf` DataFrame, where `value` is smaller than 1000, and, the `country` is Brazil, and, `user` is Eduardo. These conditions are dependent, that is, they are connected to each other. That is why I used the `and` keyword in the example below, to connect these three conditions together.

```
condition = '''
  value < 1000 and country == 'Brazil' and user == 'Eduardo'
'''

transf\
  .filter(condition)\
  .show(5)
```

```
+-------------------+-------+--------+----------+-------+-----------+
|           datetime|   user|   value|transferid|country|description|
+-------------------+-------+--------+----------+-------+-----------+
|2018-12-06 12:00:47|Eduardo|74.54787| 114785689| Brazil|       null|
|2018-12-05 18:55:00|Eduardo| 422.832| 115120371| Brazil|       null|
|2018-12-05 14:12:52|Eduardo|23.65436| 114706389| Brazil|       null|
|2018-12-05 12:59:46|Eduardo|729.4454| 115857204| Brazil|       null|
|2018-12-04 17:51:20|Eduardo|416.6055| 116252992| Brazil|       null|
+-------------------+-------+--------+----------+-------+-----------+
```

```
only showing top 5 rows
```

I could translate this logical expression into the "pythonic" way (using the `col()` function). However, I would have to surround each individual expression by parentheses, and, use the `&` operator to substitute the `and` keyword.

```
transf\
  .filter(
    (col('value') < 1000) & (col('country') == 'Brazil') & (col('user') == 'Eduardo')
  )\
  .show(5)
```

```
+-------------------+-------+--------+----------+-------+-----------+
|           datetime|   user|   value|transferid|country|description|
+-------------------+-------+--------+----------+-------+-----------+
|2018-12-06 12:00:47|Eduardo|74.54787| 114785689| Brazil|       null|
|2018-12-05 18:55:00|Eduardo| 422.832| 115120371| Brazil|       null|
|2018-12-05 14:12:52|Eduardo|23.65436| 114706389| Brazil|       null|
|2018-12-05 12:59:46|Eduardo|729.4454| 115857204| Brazil|       null|
|2018-12-04 17:51:20|Eduardo|416.6055| 116252992| Brazil|       null|
+-------------------+-------+--------+----------+-------+-----------+
only showing top 5 rows
```

This a very important detail, because is very easy to forget it. When building your complex logical expressions, if you are using the `col()` function, always remember to surround each expression by a pair of parentheses. Otherwise, you will get a very confusing and useless error message, like this:

```
transf\
  .filter(
    col('value') < 1000 & col('country') == 'Brazil' & col('user') == 'Eduardo'
  )\
  .show(5)
```

```
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
  File "/usr/local/spark/python/pyspark/sql/column.py", line 111, in _
    njc = getattr(self._jc, name)(jc)
  File "/usr/local/spark/python/lib/py4j-0.10.9.3-src.zip/py4j/java_gateway.py", line 1321
  File "/usr/local/spark/python/pyspark/sql/utils.py", line 111, in deco
    return f(*a, **kw)
```

```
  File "/usr/local/spark/python/lib/py4j-0.10.9.3-src.zip/py4j/protocol.py", line 330, in
py4j.protocol.Py4JError: An error occurred while calling o89.and. Trace:
py4j.Py4JException: Method and([class java.lang.Integer]) does not exist
    at py4j.reflection.ReflectionEngine.getMethod(ReflectionEngine.java:318)
    at py4j.reflection.ReflectionEngine.getMethod(ReflectionEngine.java:326)
    at py4j.Gateway.invoke(Gateway.java:274)
    at py4j.commands.AbstractCommand.invokeMethod(AbstractCommand.java:132)
    at py4j.commands.CallCommand.execute(CallCommand.java:79)
    at py4j.ClientServerConnection.waitForCommands(ClientServerConnection.java:182)
    at py4j.ClientServerConnection.run(ClientServerConnection.java:106)
    at java.lang.Thread.run(Thread.java:748)
```

In the above examples, we have logical expressions that are dependent on each other. But, lets suppose these conditions were independent. In this case, we would use the `or` keyword, instead of `and`. Now, Spark will look for every row of `transf` where `value` is smaller than 1000, or, `country` is Brazil, or, `user` is Eduardo.

```
condition = '''
  value < 1000 or country == 'Brazil' or user == 'Eduardo'
'''

transf\
  .filter(condition)\
  .show(5)
```

```
+-------------------+-----------+--------+----------+-------+-----------+
|           datetime|       user|   value|transferid|country|description|
+-------------------+-----------+--------+----------+-------+-----------+
|2018-12-06 20:19:19|    Eduardo|598.5984| 116241629|Germany|       null|
|2018-12-06 18:54:32|    Eduardo|5665.214| 114830203|Germany|       null|
|2018-12-06 17:07:50|    Eduardo|9560.668| 115917812|Germany|       null|
|2018-12-06 16:09:15|Júlio Cesar| 387.596| 114894102|Germany|       null|
|2018-12-06 14:59:38|    Eduardo|11758.72| 115580064|Germany|       null|
+-------------------+-----------+--------+----------+-------+-----------+
only showing top 5 rows
```

To translate this expression into the pythonic way, we have to substitute the `or` keyword by the `|` operator, and surround each expression by parentheses again:

```
transf\
  .filter(
    (col('value') < 1000) | (col('country') == 'Brazil') | (col('user') == 'Eduardo')
  )\
  .show(5)
```

```
+-------------------+-----------+--------+----------+-------+-----------+
|           datetime|       user|   value|transferid|country|description|
+-------------------+-----------+--------+----------+-------+-----------+
|2018-12-06 20:19:19|    Eduardo|598.5984| 116241629|Germany|       null|
|2018-12-06 18:54:32|    Eduardo|5665.214| 114830203|Germany|       null|
|2018-12-06 17:07:50|    Eduardo|9560.668| 115917812|Germany|       null|
|2018-12-06 16:09:15|Júlio Cesar| 387.596| 114894102|Germany|       null|
|2018-12-06 14:59:38|    Eduardo|11758.72| 115580064|Germany|       null|
+-------------------+-----------+--------+----------+-------+-----------+
only showing top 5 rows
```

You can increase the complexity of your logical expressions by mixing dependent expressions with independent expressions. For example, to filter all the rows where `country` is Germany, and, `user` is either Eduardo or Ana, you would have the following code:

```
condition = '''
  (user == 'Eduardo' or user == 'Ana') and country == 'Germany'
'''

transf\
  .filter(condition)\
  .show(5)
```

```
+-------------------+-------+--------+----------+-------+-----------+
|           datetime|   user|   value|transferid|country|description|
+-------------------+-------+--------+----------+-------+-----------+
|2018-12-06 20:19:19|Eduardo|598.5984| 116241629|Germany|       null|
|2018-12-06 19:41:27|    Ana|1408.261| 116262934|Germany|       null|
|2018-12-06 18:54:32|Eduardo|5665.214| 114830203|Germany|       null|
|2018-12-06 17:48:39|    Ana|4588.665| 116281690|Germany|       null|
|2018-12-06 17:07:50|Eduardo|9560.668| 115917812|Germany|       null|
+-------------------+-------+--------+----------+-------+-----------+
only showing top 5 rows
```

If you investigate the above condition carefully, maybe, you will identify that this condition could be rewritten in a simpler format, by using the `in` keyword. This way, Spark will look for all the rows where `user` is equal to one of the listed values (Eduardo and Ana), and, that `country` is Germany.

```
condition = '''
  user in ('Eduardo', 'Ana') and country == 'Germany'
'''
```

4.5.3 Translating the `in` keyword to the pythonic way

Python does have a `in` keyword just like SQL, but, this keyword does not work as expected in `pyspark`. To write a logical expression, using the pythonic way, that filters the rows where a column is equal to one of the listed values, you can use the `isin()` method.

This method belongs to the `Column` class, so, you should always use `isin()` after a column name or a `col()` function. In the example below, we are filtering the rows where `user` is Eduardo or Júlio Cesar:

```
transf\
  .filter(col('user').isin('Eduardo', 'Júlio Cesar'))\
  .show(5)
```

```
+-------------------+-----------+--------+----------+-------+-----------+
|           datetime|       user|   value|transferid|country|description|
+-------------------+-----------+--------+----------+-------+-----------+
|2018-12-06 20:19:19|    Eduardo|598.5984| 116241629|Germany|       null|
|2018-12-06 18:54:32|    Eduardo|5665.214| 114830203|Germany|       null|
|2018-12-06 17:07:50|    Eduardo|9560.668| 115917812|Germany|       null|
|2018-12-21 16:46:59|Júlio Cesar|16226.42| 116279014|Germany|       null|
|2018-12-06 16:09:15|Júlio Cesar| 387.596| 114894102|Germany|       null|
+-------------------+-----------+--------+----------+-------+-----------+
only showing top 5 rows
```

4.5.4 Negating logical conditions

In some cases, is easier to describe what rows you **do not want** in your filter. In this case, you want to negate (or invert) your logical expression. For this, SQL provides the `not` keyword, that you place before the logical expression you want to negate.

For example, we can filter all the rows of `transf` where `user` is not equal to Ana. Remember, the methods `filter()` and `where()` are equivalents or synonymous (they both mean the same thing).

```
condition = '''
  not user == 'Ana'
'''

transf\
  .where(condition)\
  .show(5)
```

```
+-------------------+--------+--------+----------+-------+-----------+
|           datetime|    user|   value|transferid|country|description|
+-------------------+--------+--------+----------+-------+-----------+
|2018-12-06 20:19:19| Eduardo|598.5984| 116241629|Germany|       null|
|2018-12-06 20:10:34|   Júlio|4610.955| 115586504|Germany|       null|
|2018-12-06 19:59:50|Nathália|4417.866| 115079280|Germany|       null|
|2018-12-06 19:54:13|   Júlio|2739.618| 114972398|Germany|       null|
|2018-12-06 19:18:40|Nathália|5051.828| 115710402|Germany|       null|
+-------------------+--------+--------+----------+-------+-----------+
only showing top 5 rows
```

To translate this expression to the pythonic way, we use the ~ operator. Because we are negating the logical expression as a whole, is important to surround the entire expression with parentheses.

```
transf\
  .where(~(col('user') == 'Ana'))\
  .show(5)
```

```
+-------------------+--------+--------+----------+-------+-----------+
|           datetime|    user|   value|transferid|country|description|
+-------------------+--------+--------+----------+-------+-----------+
|2018-12-06 20:19:19| Eduardo|598.5984| 116241629|Germany|       null|
|2018-12-06 20:10:34|   Júlio|4610.955| 115586504|Germany|       null|
|2018-12-06 19:59:50|Nathália|4417.866| 115079280|Germany|       null|
|2018-12-06 19:54:13|   Júlio|2739.618| 114972398|Germany|       null|
|2018-12-06 19:18:40|Nathália|5051.828| 115710402|Germany|       null|
+-------------------+--------+--------+----------+-------+-----------+
only showing top 5 rows
```

If you forget to do this, Spark will think you are negating just the column (e.g. `~col('user')`), and not the entire expression. That would not make sense, and, as a result, Spark would throw an error:

```
transf\
  .where(~col('user') == 'Ana')\
  .show(5)
```

```
AnalysisException: cannot resolve '(NOT user)' due to data type mismatch: argument 1 requi
'Filter (NOT user#25 = Ana)
```

Because the `~` operator is a little discrete and can go unnoticed, I sometimes use a different approach to negate my logical expressions. I make the entire expression equal to `False`. This way, I get all the rows where that particular expression is `False`. This makes my intention more visible in the code, but, is harder to write it.

```
# Filter all the rows where `user` is not equal to
# Ana or Eduardo.
transf\
  .where( (col('user').isin('Ana', 'Eduardo')) == False )\
  .show(5)
```

```
+-------------------+--------+--------+----------+-------+-----------+
|           datetime|    user|   value|transferid|country|description|
+-------------------+--------+--------+----------+-------+-----------+
|2018-12-06 20:10:34|   Júlio|4610.955| 115586504|Germany|       null|
|2018-12-06 19:59:50|Nathália|4417.866| 115079280|Germany|       null|
|2018-12-06 19:54:13|   Júlio|2739.618| 114972398|Germany|       null|
|2018-12-06 19:18:40|Nathália|5051.828| 115710402|Germany|       null|
|2018-12-06 18:15:46|  Sandra| 1474.44| 116323455|Germany|       null|
+-------------------+--------+--------+----------+-------+-----------+
only showing top 5 rows
```

4.5.5 Filtering `null` values

Sometimes, the `null` values play an important role in your filter. You either want to collect all these `null` values, so you can investigate why they are null in the first place, or, you want to completely eliminate them from your DataFrame.

Because this is a special kind of value in Spark, with a special meaning (the "absence" of a value), you need to use a special syntax to correctly filter these values in your DataFrame. In SQL, you can use the `is` keyword to filter these values:

```
transf\
  .where('user is null')\
  .show()
```

```
+-------------------+----+-------+----------+-------+-----------+
|           datetime|user|  value|transferid|country|description|
+-------------------+----+-------+----------+-------+-----------+
|2018-11-10 02:02:56|null|   null| 166420322|   null|       null|
|2018-11-10 11:12:06|null| 170.21| 166420323|   null|       null|
|2018-11-10 03:12:09|null|1002.35| 166420324|   null|       null|
|2018-11-10 00:43:41|null|2560.76| 166420325|   null|       null|
+-------------------+----+-------+----------+-------+-----------+
```

However, if you want to remove these values from your DataFrame, then, you can just negate (or invert) the above expression with the `not` keyword, like this:

```
transf\
  .where('not user is null')\
  .show(5)
```

```
+-------------------+--------+--------+----------+-------+-----------+
|           datetime|    user|   value|transferid|country|description|
+-------------------+--------+--------+----------+-------+-----------+
|2018-12-06 20:19:19| Eduardo|598.5984| 116241629|Germany|       null|
|2018-12-06 20:10:34|   Júlio|4610.955| 115586504|Germany|       null|
|2018-12-06 19:59:50|Nathália|4417.866| 115079280|Germany|       null|
|2018-12-06 19:54:13|   Júlio|2739.618| 114972398|Germany|       null|
|2018-12-06 19:41:27|     Ana|1408.261| 116262934|Germany|       null|
+-------------------+--------+--------+----------+-------+-----------+
only showing top 5 rows
```

The `is` and `not` keywords have a special relation. Because you can create the same negation/inversion of the expression by inserting the `not` keyword in the middle of the expression (you can do this too in expressions with the `in` keyword). In other words, you might see in someone else's code this expression written in this form:

```
transf\
  .where('user is not null')\
  .show(5)
```

```
+-------------------+--------+--------+----------+-------+-----------+
|           datetime|    user|   value|transferid|country|description|
+-------------------+--------+--------+----------+-------+-----------+
|2018-12-06 20:19:19| Eduardo|598.5984| 116241629|Germany|       null|
|2018-12-06 20:10:34|   Júlio|4610.955| 115586504|Germany|       null|
|2018-12-06 19:59:50|Nathália|4417.866| 115079280|Germany|       null|
|2018-12-06 19:54:13|   Júlio|2739.618| 114972398|Germany|       null|
|2018-12-06 19:41:27|     Ana|1408.261| 116262934|Germany|       null|
+-------------------+--------+--------+----------+-------+-----------+
only showing top 5 rows
```

This is a valid SQL logical expression, but is a strange one. Because we cannot use the `not` keyword in this manner on other kinds of logical expressions. Normally, we put the `not` keyword **before** the logical expression we want to negate, not in the middle of it. Anyway, just have in mind that this form of logical expression exists, and, that is a perfectly valid one.

When we translate the above examples to the "pythonic" way, many people tend to use the `null` equivalent of python, that is, the `None` value, in the expression. But as you can see in the result below, this method does not work as expected:

```
transf\
  .where(col('user') == None)\
  .show()
```

```
+--------+----+-----+----------+-------+-----------+
|datetime|user|value|transferid|country|description|
+--------+----+-----+----------+-------+-----------+
+--------+----+-----+----------+-------+-----------+
```

The correct way to do this in `pyspark`, is to use the `isNull()` method from the `Column` class.

```
transf\
  .where(col('user').isNull())\
  .show()
```

```
+-------------------+----+-------+----------+-------+-----------+
|           datetime|user|  value|transferid|country|description|
+-------------------+----+-------+----------+-------+-----------+
|2018-11-10 02:02:56|null|   null| 166420322|   null|       null|
|2018-11-10 11:12:06|null| 170.21| 166420323|   null|       null|
|2018-11-10 03:12:09|null|1002.35| 166420324|   null|       null|
|2018-11-10 00:43:41|null|2560.76| 166420325|   null|       null|
+-------------------+----+-------+----------+-------+-----------+
```

If you want to eliminate the `null` values, just use the inverse method `isNotNull()`.

```
transf\
  .where(col('user').isNotNull())\
  .show(5)
```

```
+-------------------+--------+--------+----------+-------+-----------+
|           datetime|    user|   value|transferid|country|description|
+-------------------+--------+--------+----------+-------+-----------+
|2018-12-06 20:19:19| Eduardo|598.5984| 116241629|Germany|       null|
|2018-12-06 20:10:34|   Júlio|4610.955| 115586504|Germany|       null|
|2018-12-06 19:59:50|Nathália|4417.866| 115079280|Germany|       null|
|2018-12-06 19:54:13|   Júlio|2739.618| 114972398|Germany|       null|
|2018-12-06 19:41:27|     Ana|1408.261| 116262934|Germany|       null|
+-------------------+--------+--------+----------+-------+-----------+
only showing top 5 rows
```

4.6 Selecting specific columns of your DataFrame

Sometimes, you need manage or transform the columns you have. For example, you might need to change the order of these columns, or, to delete/rename some of them. To do this, you can use the `select()` and `drop()` methods of your DataFrame.

The `select()` method works very similarly to the `SELECT` statement of SQL. You basically list all the columns you want to keep in your DataFrame, in the specific order you want.

```
transf\
  .select('transferid', 'datetime', 'user', 'value')\
  .show(5)
```

```
+----------+-------------------+--------+--------+
|transferid|           datetime|    user|   value|
+----------+-------------------+--------+--------+
| 116241629|2018-12-06 20:19:19| Eduardo|598.5984|
| 115586504|2018-12-06 20:10:34|   Júlio|4610.955|
| 115079280|2018-12-06 19:59:50|Nathália|4417.866|
| 114972398|2018-12-06 19:54:13|   Júlio|2739.618|
| 116262934|2018-12-06 19:41:27|     Ana|1408.261|
+----------+-------------------+--------+--------+
only showing top 5 rows
```

4.6.1 Renaming your columns

Realize in the example above, that the column names can be delivered directly as strings to `select()`. This makes life pretty easy, but, it does not give you extra options. You may want to rename some of the columns, and, to do this, you need to use the `alias()` method from `Column` class. Since this is a method from `Column` class, I always use it after a `col()` function, or, after a column name.

```
transf\
  .select(
    'datetime',
    col('transferid').alias('transferID'),
    transf.user.alias('clientName')
  )\
  .show(5)
```

```
+-------------------+----------+----------+
|           datetime|transferID|clientName|
+-------------------+----------+----------+
|2018-12-06 20:19:19| 116241629|   Eduardo|
|2018-12-06 20:10:34| 115586504|     Júlio|
|2018-12-06 19:59:50| 115079280|  Nathália|
|2018-12-06 19:54:13| 114972398|     Júlio|
|2018-12-06 19:41:27| 116262934|       Ana|
+-------------------+----------+----------+
only showing top 5 rows
```

4.6.2 Dropping unnecessary columns

In some cases, your DataFrame just have too many columns and you just want to eliminate a few of them. In a situation like this, you can list the columns you want to drop from your DataFrame, inside the `drop()` method, like this:

```
transf\
  .drop('user', 'description')\
  .show(5)
```

```
+-------------------+--------+----------+-------+
|           datetime|   value|transferid|country|
+-------------------+--------+----------+-------+
|2018-12-06 20:19:19|598.5984| 116241629|Germany|
|2018-12-06 20:10:34|4610.955| 115586504|Germany|
|2018-12-06 19:59:50|4417.866| 115079280|Germany|
|2018-12-06 19:54:13|2739.618| 114972398|Germany|
|2018-12-06 19:41:27|1408.261| 116262934|Germany|
+-------------------+--------+----------+-------+
only showing top 5 rows
```

4.6.3 You can add new columns with `select()`

When I said that `select()` works in the same way as the `SELECT` statement of SQL, I also meant that you can use `select()` to select columns that do not currently exist in your DataFrame, and add them to the final result.

For example, I can select a new column (called `by_1000`) containing `value` divided by 1000, like this:

```
transf\
  .select(
    'value',
    (col('value') / 1000).alias('by_1000')
  )\
  .show(5)
```

```
+--------+---------+
|   value|  by_1000|
+--------+---------+
|598.5984|0.5985984|
```

```
|4610.955| 4.610955|
|4417.866| 4.417866|
|2739.618| 2.739618|
|1408.261| 1.408261|
+--------+---------+
only showing top 5 rows
```

So this `by_1000` column do not exist in `transf` DataFrame, it was calculated and added to the final result by `select()`. The formula `col('value') / 1000` is the equation that defines what this `by_1000` column is, or, how it should be calculated.

Besides that, `select()` provides a useful shortcut to reference all the columns of your DataFrame. That is, the star symbol (`*`) from the `SELECT` statement in SQL. This shortcut is very useful when you want to maintain all columns, and, add a new column, at the same time.

In the example below, we are adding the same `by_1000` column, however, we are bringing all the columns of `transf` together.

```
transf\
  .select(
    '*',
    (col('value') / 1000).alias('by_1000')
  )\
  .show(5)
```

```
+-------------------+--------+--------+----------+-------+-----------+---------+
|           datetime|    user|   value|transferid|country|description|  by_1000|
+-------------------+--------+--------+----------+-------+-----------+---------+
|2018-12-06 20:19:19| Eduardo|598.5984| 116241629|Germany|       null|0.5985984|
|2018-12-06 20:10:34|   Júlio|4610.955| 115586504|Germany|       null| 4.610955|
|2018-12-06 19:59:50|Nathália|4417.866| 115079280|Germany|       null| 4.417866|
|2018-12-06 19:54:13|   Júlio|2739.618| 114972398|Germany|       null| 2.739618|
|2018-12-06 19:41:27|     Ana|1408.261| 116262934|Germany|       null| 1.408261|
+-------------------+--------+--------+----------+-------+-----------+---------+
only showing top 5 rows
```

4.6.4 Casting columns to a different data type

Spark try to do its best when guessing which is correct data type for the columns of your DataFrame. But, obviously, Spark can get it wrong, and, you end up deciding by your own which data type to use for a specific column.

To explicit transform to, or, define which is the correct data type of a column, you can use the `cast()` method inside `select()`. This `cast()` method belongs to the `Column` class, so, you should always use it after a column name, or, a `col()` function:

```
transf\
  .select(
    'value',
    col('value').cast('long').alias('value_as_integer'),
    transf.value.cast('boolean').alias('value_as_boolean')
  )\
  .show(5)
```

```
+--------+----------------+----------------+
|   value|value_as_integer|value_as_boolean|
+--------+----------------+----------------+
|598.5984|             598|            true|
|4610.955|            4610|            true|
|4417.866|            4417|            true|
|2739.618|            2739|            true|
|1408.261|            1408|            true|
+--------+----------------+----------------+
only showing top 5 rows
```

To use this `cast()` method, you give the name of the data type (as a string) to which you want to cast the column. The main available data types to `cast()` are:

- `'string'`: correspond to `StringType()`;
- `'int'`: correspond to `IntegerType()`;
- `'long'`: correspond to `LongType()`;
- `'double'`: correspond to `DoubleType()`;
- `'date'`: correspond to `DateType()`;
- `'timestamp'`: correspond to `TimestampType()`;
- `'boolean'`: correspond to `BooleanType()`;
- `'array'`: correspond to `ArrayType()`;
- `'dict'`: correspond to `MapType()`;

4.7 Calculating or adding new columns to your DataFrame

Although you can add new columns with `select()`, this method is not specialized to do that. As consequence, when you want to add many new columns, it can become pretty annoying

to write `select('*', new_column)` over and over again. That is why `pyspark` provides a special method called `withColumn()`.

This method has two arguments. First, is the name of the new column. Second, is the formula or equation that represents this new column. As an example, I could reproduce the same `by_1000` column like this:

```
transf\
  .withColumn('by_1000', col('value') / 1000)\
  .show(5)
```

```
+-------------------+--------+--------+----------+-------+-----------+---------+
|           datetime|    user|   value|transferid|country|description|  by_1000|
+-------------------+--------+--------+----------+-------+-----------+---------+
|2018-12-06 20:19:19| Eduardo|598.5984| 116241629|Germany|       null|0.5985984|
|2018-12-06 20:10:34|   Júlio|4610.955| 115586504|Germany|       null| 4.610955|
|2018-12-06 19:59:50|Nathália|4417.866| 115079280|Germany|       null| 4.417866|
|2018-12-06 19:54:13|   Júlio|2739.618| 114972398|Germany|       null| 2.739618|
|2018-12-06 19:41:27|     Ana|1408.261| 116262934|Germany|       null| 1.408261|
+-------------------+--------+--------+----------+-------+-----------+---------+
only showing top 5 rows
```

You will see a lot more examples of formulas and uses of `withColumn()` throughout this book. I just want you to know that `withColumn()` is a method that adds a new column to your DataFrame. The first argument is the name of the new column, and, the second argument is the calculation formula of this new column.

4.8 Sorting rows of your DataFrame

Spark, or, `pyspark`, provides the `orderBy()` DataFrame method to sort rows. To use this method, you give the name of the columns that Spark will use to sort the rows.

In the example below, Spark will sort `transf` according to the values in the `value` column. By default, Spark uses an ascending order while sorting your rows.

```
transf\
  .orderBy('value')\
  .show(5)
```

```
+-------------------+--------+---------+----------+---------+-----------+
|           datetime|    user|    value|transferid|  country|description|
+-------------------+--------+---------+----------+---------+-----------+
|2018-11-10 02:02:56|    null|     null| 166420322|     null|       null|
|2018-11-21 19:07:26|Nathália|0.2619445| 115194733|Venezuela|       null|
|2018-11-25 21:29:42|  Sandra|0.2996283| 114302779|Argentina|       null|
|2018-11-14 18:42:04|nathalia|0.6452489| 115959375|  Ecuador|       null|
|2018-11-21 16:31:12| Armando|0.6727713| 114940976|Venezuela|       null|
+-------------------+--------+---------+----------+---------+-----------+
only showing top 5 rows
```

Just to be clear, you can use the combination between multiple columns to sort your rows. Just give the name of each column (as strings) separated by commas. In the example below, Spark will sort the rows according to `user` column first, then, is going to sort the rows of each `user` according to `value` column.

```
transf\
  .orderBy('user', 'value')\
  .show(5)
```

```
+-------------------+----+--------+----------+---------+-----------+
|           datetime|user|   value|transferid|  country|description|
+-------------------+----+--------+----------+---------+-----------+
|2018-11-10 02:02:56|null|    null| 166420322|     null|       null|
|2018-11-10 11:12:06|null|  170.21| 166420323|     null|       null|
|2018-11-10 03:12:09|null| 1002.35| 166420324|     null|       null|
|2018-11-10 00:43:41|null| 2560.76| 166420325|     null|       null|
|2018-11-30 09:45:35| Ana|3.581695| 114564798|Venezuela|       null|
+-------------------+----+--------+----------+---------+-----------+
only showing top 5 rows
```

If you want to use a descending order in a specific column, you need to use the `desc()` method from `Column` class. In the example below, Spark will sort the rows according to `user` column using an ascending order. However, it will use the values from `value` column in a descending order to sort the rows in each `user`.

```
transf\
  .orderBy('user', col('value').desc())\
  .show(5)
```

```
+-------------------+----+--------+----------+-------+-----------+
|           datetime|user|   value|transferid|country|description|
+-------------------+----+--------+----------+-------+-----------+
|2018-11-10 00:43:41|null| 2560.76| 166420325|   null|       null|
|2018-11-10 03:12:09|null| 1002.35| 166420324|   null|       null|
|2018-11-10 11:12:06|null|  170.21| 166420323|   null|       null|
|2018-11-10 02:02:56|null|    null| 166420322|   null|       null|
|2018-11-16 06:57:56| Ana|35129.43| 115902710|Ecuador|       null|
+-------------------+----+--------+----------+-------+-----------+
only showing top 5 rows
```

This means that you can mix ascending orders with descending orders in `orderBy()`. Since the ascending order is the default, if you want to use a descending order in all of the columns, then, you need to apply the `desc()` method to all of them.

5 Working with SQL in Spark DataFrames

6 Importing data to Spark

6.1 Importing data from different data sources

Another way of creating Spark DataFrames, is to read (or import) data from a file and convert it to a DataFrame. Spark can read a variety of file formats, including CSV, Parquet, JSON, ORC and Binary files. Furthermore, Spark can connect to other databases and import tables from them, using JDBC connections. We can access the read engines for these different file formats, by using the `read` module from your Spark Session object.

6.1.1 Reading data from static files

Static files are probably the easiest way to transport data from one computer to another. Because you just need to copy and paste this file to this other machine, or download it from the internet. To read any file stored inside your computer, Spark always need to know the path to this file.

As an example, I have the following CSV file saved in my computer:

```
name,age,job
Jorge,30,Developer
Bob,32,Developer
```

This CSV was saved in a file called `people.csv`, inside a folder called `Data`. So, to read this file, I gave the path to this CSV file to the `read.csv()` method of my Spark Session, like in the example below:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

path = "../Data/people.csv"

df = spark.read.csv(path)
df.show()
```

```
22/06/14 19:11:51 WARN Utils: Your hostname, pedro-IdeaPad-3-15ALC6 resolves to a loopback ad
22/06/14 19:11:51 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another address

Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).

22/06/14 19:11:52 WARN NativeCodeLoader: Unable to load native-hadoop library for your platfo

+-----+---+---------+
|  _c0|_c1|      _c2|
+-----+---+---------+
| name|age|      job|
|Jorge| 30|Developer|
|  Bob| 32|Developer|
+-----+---+---------+
```

In the above example, I gave a relative path to the file I wanted to read. But you can provide an absolute path[1] to the file, if you want to.

The `people.csv` is located at a very specific folder in my Linux computer, so, the absolute path to this file is pretty long as you can see below. But, if I were in my Windows machine, this absolute path would be something like `"C:\Users\pedro\Documents\Projects\..."`.

```
path = "/home/pedro/Documents/Projets/Books/Introd-pyspark/Data/people.csv"

df = spark.read.csv(path)
df.show()
```

```
+-----+---+---------+
|  _c0|_c1|      _c2|
+-----+---+---------+
| name|age|      job|
|Jorge| 30|Developer|
|  Bob| 32|Developer|
+-----+---+---------+
```

[1] That is, the complete path to the file, or, in other words, a path that starts in the root folder of your hard drive.

If you give an invalid path (that is, a path that does not exist in your computer), you will get a `AnalysisException`. In the example below, I try to read a file called `"weird-file.csv"` that (in theory) is located at my current working directory. But when Spark looks inside my current directory, it does not find any file called `"weird-file.csv"`. As a result, Spark raises a `AnalysisException` that warns me about this mistake.

```
df = spark.read.csv("weird-file.csv")
```

```
Traceback (most recent call last):
  ...
pyspark.sql.utils.AnalysisException: Path does not exist: file:/home/pedro/Documents/Proje
```

Even CSV's being a very popular format, is very likely that you will need to read archives in many different formats. The main read engines available in Spark for static files are listed below:

- `spark.read.json()`: to read JSON files;
- `spark.read.csv()`: to read CSV files;
- `spark.read.parquet()`: to read Apache Parquet files;
- `spark.read.orc()`: to read ORC (Apache *Optimized Row Columnar* format) files;

6.1.2 Defining import options

While reading and importing your files, Spark will use the default values for the import options defined by the read engine you are using, unless you explicit ask it to use different values. Each read engine has its own read/import options.

For example, the `spark.read.orc()` engine has a option called `mergeSchema`. With this option, you can ask Spark to merge the schemas collected from all the ORC part-files. In contrast, the `spark.read.csv()` engine does not have such option. Because this functionality of "merging schemas" does not make sense with CSV files.

This means that, some import options are specific (or characteristic) of some file formats. For example, the `sep` option (where you define the *separator* character) is used only in the `spark.read.csv()` engine, because you do not have a special character that behaves as the "separator" in the other file formats (ORC, JSON, Parquet…). So it does not make sense to have such option in the other read engines.

In the other hand, some import options can co-exist in multiple read engines. For example, the `spark.read.json()` and `spark.read.csv()` have both an `encoding` option. The encoding is a very important information, and Spark needs it to correct interpret your file. By default, Spark will always assume that your files use the UTF-8 encoding system. Although this may

not be true for your specific case, and for these cases you use this `encoding` option to tell Spark which one to use.

In the next sections, I will break down some of the most used import options for each file format. If you want to see the complete list of import options, you can visit the *Data Source Option* section in the specific part of the file format you are using in the Spark SQL Guide[2].

6.1.3 Import options for CSV files

The most important import options for CSV files are `sep`, `encoding`, `header`.

6.1.4 Pulling data from SQL Databases

We can use the `spark.read.jdbc()` method to connect and read data from Databases using JDBC connections. Also, you can read a SQL table from your Spark context by using the `spark.table()` method.

[2]For example, this *Data Source Option* for Parque files is located at: https://spark.apache.org/docs/latest/sql-data-sources-parquet.html#data-source-option

References

Apache Spark Official Documentation. 2022. Documentation for Apache Spark 3.2.1; Available at: https://spark.apache.org/docs/latest/.

Chambers, Bill, and Matei Zaharia. 2018. *Spark: The Definitive Guide: Big Data Processing Made Simple*. Sebastopol, CA: O'Reilly Media.

Karau, Holden, Andy Konwinski, Patrick Wendell, and Matei Zaharia. 2015. *Learning Spark*. Sebastopol, CA: O'Reilly Media.