

# **Introduction to pyspark**

Pedro Duarte Faria

2023-12-19

# Table of contents

<b>Welcome</b>	<b>7</b>
<b>Preface</b>	<b>8</b>
About this book . . . . .	8
About the author . . . . .	9
Some conventions of this book . . . . .	9
Python code and terminal commands . . . . .	9
Python objects, functions and methods . . . . .	10
Be aware of differences between OS's! . . . . .	10
Install the necessary software . . . . .	11
Book's metadata . . . . .	11
License . . . . .	11
Book citation . . . . .	11
Corresponding author and maintainer . . . . .	11
<b>1 Key concepts of python</b>	<b>12</b>
1.1 Scripts . . . . .	12
1.2 How to run a python program . . . . .	12
1.3 Objects . . . . .	13
1.4 Expressions . . . . .	15
1.5 Packages (or libraries) . . . . .	17
1.6 Methods versus Functions . . . . .	20
1.7 Identifying classes and their methods . . . . .	22
<b>2 Introducing Apache Spark</b>	<b>23</b>
2.1 What is Spark? . . . . .	23
2.2 Spark application . . . . .	24
2.3 Spark application versus pyspark application . . . . .	25
2.4 Core parts of a pyspark program . . . . .	26
2.4.1 Importing the pyspark package (or modules) . . . . .	26
2.4.2 Starting your Spark Session . . . . .	26
2.4.3 Defining a set of transformations and actions . . . . .	27
2.5 Building your first Spark application . . . . .	27
2.5.1 Writing the code . . . . .	27
2.5.2 Executing the code . . . . .	28

2.6	Overview of pyspark . . . . .	29
2.6.1	Main python modules . . . . .	29
2.6.2	Main python classes . . . . .	30
<b>3</b>	<b>Introducing Spark DataFrames</b>	<b>31</b>
3.1	Spark DataFrames versus Spark Datasets . . . . .	31
3.2	Partitions of a Spark DataFrame . . . . .	32
3.3	The DataFrame class in pyspark . . . . .	33
3.4	Building a Spark DataFrame . . . . .	35
3.5	Visualizing a Spark DataFrame . . . . .	37
3.6	Getting the name of the columns . . . . .	38
3.7	Getting the number of rows . . . . .	39
3.8	Spark Data Types . . . . .	39
3.9	The DataFrame Schema . . . . .	40
3.9.1	Accessing the DataFrame schema . . . . .	41
3.9.2	Building a DataFrame schema . . . . .	42
3.9.3	Checking your DataFrame schema . . . . .	43
<b>4</b>	<b>Introducing the Column class</b>	<b>45</b>
4.1	Building a column object . . . . .	45
4.2	Columns are strongly related to expressions . . . . .	46
4.3	Literal values versus expressions . . . . .	47
4.4	Passing a literal (or a constant) value to Spark . . . . .	48
4.5	Key methods of the Column class . . . . .	49
<b>5</b>	<b>Transforming your Spark DataFrame - Part 1</b>	<b>51</b>
5.1	Defining transformations . . . . .	51
5.2	Triggering calculations with actions . . . . .	53
5.3	Understanding narrow and wide transformations . . . . .	54
5.4	The transf DataFrame . . . . .	56
5.5	Filtering rows of your DataFrame . . . . .	60
5.5.1	Logical operators available . . . . .	61
5.5.2	Connecting multiple logical expressions . . . . .	62
5.5.3	Translating the in keyword to the pythonic way . . . . .	67
5.5.4	Negating logical conditions . . . . .	67
5.5.5	Filtering null values (i.e. missing data) . . . . .	70
5.5.6	Filtering dates and datetimes in your DataFrame . . . . .	72
5.5.7	Searching for a particular pattern in string values . . . . .	74
5.6	Selecting a subset of rows from your DataFrame . . . . .	77
5.6.1	Limiting the number of rows in your DataFrame . . . . .	77
5.6.2	Getting the first/last $n$ rows of your DataFrame . . . . .	77
5.6.3	Taking a random sample of your DataFrame . . . . .	78

5.7	Managing the columns of your DataFrame . . . . .	80
5.7.1	Renaming your columns . . . . .	80
5.7.2	Dropping unnecessary columns . . . . .	81
5.7.3	Casting columns to a different data type . . . . .	82
5.7.4	You can add new columns with select() . . . . .	83
5.8	Calculating or adding new columns to your DataFrame . . . . .	84
5.9	Sorting rows of your DataFrame . . . . .	85
5.10	Calculating aggregates . . . . .	87
5.10.1	Using standard DataFrame methods . . . . .	87
5.10.2	Using the agg() method . . . . .	88
5.10.3	Without groups, we calculate a aggregate of the entire DataFrame . . . . .	89
5.10.4	Calculating aggregates per group in your DataFrame . . . . .	89
<b>6</b>	<b>Importing data to Spark</b>	<b>92</b>
6.1	Reading data from static files . . . . .	92
6.2	An example with a CSV file . . . . .	93
6.3	Import options . . . . .	95
6.4	Setting the separator character for CSV files . . . . .	97
6.5	Setting the encoding of the file . . . . .	98
6.6	Setting the format of dates and timestamps . . . . .	99
<b>7</b>	<b>Working with SQL in pyspark</b>	<b>101</b>
7.1	The sql() method as the main entrypoint . . . . .	101
7.1.1	A single SQL statement per run . . . . .	102
7.2	Creating SQL Tables in Spark . . . . .	105
7.2.1	TABLEs versus VIEWs . . . . .	106
7.2.2	Temporary versus Persistent sources . . . . .	112
7.2.3	Spark SQL Catalog is the bridge between SQL and pyspark . . . . .	114
7.3	The penguins DataFrame . . . . .	114
7.4	Selecting your Spark DataFrames . . . . .	115
7.5	Executing SQL expressions . . . . .	116
7.6	Every DataFrame transformation in Python can be translated into SQL . . . . .	118
7.6.1	DataFrame methods are usually translated into SQL keywords . . . . .	118
7.6.2	Spark functions are usually translated into SQL functions . . . . .	120
<b>8</b>	<b>Transforming your Spark DataFrame - Part 2</b>	<b>122</b>
8.1	Removing duplicated values from your DataFrame . . . . .	122
8.2	Other techniques for dealing with null values . . . . .	125
8.2.1	Replacing null values . . . . .	125
8.2.2	Dropping all null values . . . . .	126
8.3	Union operations . . . . .	127
8.4	Join operations . . . . .	131
8.4.1	What is a JOIN ? . . . . .	133

8.4.2	The different types of JOIN . . . . .	135
8.4.3	A cross JOIN as the seventh type . . . . .	140
8.5	Pivot operations . . . . .	142
8.5.1	Transforming columns into rows . . . . .	142
8.5.2	Transforming rows into columns . . . . .	146
8.6	Collecting and explode operations . . . . .	151
8.6.1	Expanding (or unnesting) with explode() . . . . .	152
8.6.2	The different versions of explode() . . . . .	154
8.6.3	Retracting (or nesting) with collect_list() and collect_set() . . . . .	155
<b>9</b>	<b>Exporting data out of Spark</b>	<b>159</b>
9.1	The write object as the main entrypoint . . . . .	159
9.2	Exporting the transf DataFrame . . . . .	160
9.2.1	Quick export to a CSV file . . . . .	160
9.2.2	Setting the write mode . . . . .	161
9.2.3	Setting write options . . . . .	163
9.3	Number of partitions determines the number of files generated . . . . .	165
9.3.1	Avoid exporting too much data into a single file . . . . .	165
9.4	Transforming to a Pandas DataFrame as a way to export data . . . . .	167
9.5	The collect() method as a way to export data . . . . .	168
<b>10</b>	<b>Tools for string manipulation</b>	<b>170</b>
10.1	The logs DataFrame . . . . .	170
10.2	Changing the case of letters in a string . . . . .	171
10.3	Calculating string length . . . . .	173
10.4	Trimming or removing spaces from strings . . . . .	173
10.5	Extracting substrings . . . . .	175
10.5.1	A substring based on a start position and length . . . . .	175
10.5.2	A substring based on a delimiter . . . . .	176
10.5.3	Forming an array of substrings . . . . .	179
10.6	Concatenating multiple strings together . . . . .	181
10.7	Introducing regular expressions . . . . .	183
10.7.1	The Java regular expression standard . . . . .	184
10.7.2	Using an invalid regular expression . . . . .	185
10.7.3	Replacing occurrences of a particular regular expression with regexp_replace() 185	
10.7.4	Introducing capturing groups on pyspark . . . . .	187
10.7.5	Extracting substrings with regexp_extract() . . . . .	190
10.7.6	Identifying values that match a particular regular expression with rlike() . . . 192	
<b>11</b>	<b>Tools for dates and datetimes manipulation</b>	<b>194</b>
11.1	Creating date values . . . . .	194
11.1.1	From strings . . . . .	194
11.1.2	From datetime values . . . . .	197

11.1.3	From individual components . . . . .	198
11.2	Creating datetime values . . . . .	199
11.2.1	From strings . . . . .	199
11.2.2	From integers . . . . .	201
11.2.3	From individual components . . . . .	203
11.3	Introducing datetime patterns . . . . .	204
11.3.1	Using datetime patterns to get date values . . . . .	205
11.3.2	Using datetime patterns to get datetime/timestamp values . . . . .	208
11.4	Extracting date or datetime components . . . . .	209
11.5	Adding time to date and datetime values with interval expressions . . . . .	210
11.6	Calculating differences between dates and datetime values . . . . .	212
11.7	Getting the now and today values . . . . .	215
<b>12</b>	<b>Introducing window functions</b>	<b>217</b>
12.1	How to define windows . . . . .	217
12.1.1	Partitioning or ordering or none . . . . .	220
12.2	Introducing the <code>over()</code> clause . . . . .	221
12.3	Window functions vs <i>group by</i> functions . . . . .	222
12.4	Ranking window functions . . . . .	223
12.5	Agreggating window functions . . . . .	227
12.6	Getting the next and previous row with <code>lead()</code> and <code>lag()</code> . . . . .	228
	<b>References</b>	<b>230</b>

**Welcome**

# Preface

## About this book

Hello! This book provides an introduction to [pyspark](#), which is a Python API to [Apache Spark](#). Here, you will learn how to perform the most common data analysis tasks and useful data transformations with Python to process huge amounts of data.

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. Since we will be using Apache Spark under the hood, it is also very important to understand a little bit of how Apache Spark works, so, we provide a small introduction to Apache 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<sup>1</sup>. Another part of the knowledge is extracted from the official documentation of Apache Spark (*Apache Spark Official Documentation* 2022), as well as some established works such as Chambers and Zaharia (2018) and Damji et al. (2020).

Some of the main subjects discussed in the book are:

- How an Apache Spark application works?
- What are Spark DataFrames?
- How to transform and model your Spark DataFrame.
- How to import data into Apache Spark.
- How to work with SQL inside pyspark.
- Tools for manipulating specific data types (e.g. string, dates and datetimes).
- How to use window functions.

---

<sup>1</sup><https://databricks.com/>



## About the author

Pedro Duarte Faria have a bachelor degree in Economics from Federal University of Ouro Preto - Brazil. Currently, he is a Data Engineer at [Blip](#), and an Associate Developer for Apache Spark 3.0 certified by Databricks.

The author have more than 3 years of experience in the data analysis market. He developed data pipelines, reports and analysis for research institutions and some of the largest companies in the brazilian financial sector, such as the BMG Bank, Sodexo and Pan Bank, besides dealing with databases that go beyond the billion rows.

Furthermore, Pedro is specialized on the R programming language, and have given several lectures and courses about it, inside graduate centers (such as PPEA-UFOP<sup>2</sup>), in addition to federal and state organizations (such as FJP-MG<sup>3</sup>). As researcher, he have experience in the field of Science, Technology and Innovation Economics.

Personal Website: <https://pedro-faria.netlify.app/>

Twitter: [@PedroPark9](#)

Mastodon: [@pedropark99@fosstodon.org](#)

## 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 written in plain black outside of the gray rectangle, just below the command that produced that visible result. 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

print(y)
```

729

---

<sup>2</sup><https://ppea.ufop.br/>

<sup>3</sup><http://fjp.mg.gov.br/>

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

```
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 Python 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.

## Install the necessary software

If you want to follow the examples shown throughout this book, you must have Apache Spark and pyspark installed on your machine. If you do not know how to do this, you can consult the [articles from phoenixNAP which are very useful](#)<sup>4</sup>.

## Book's metadata

### License

Copyright © 2024 Pedro Duarte Faria. This book is licensed by the [CC-BY 4.0 Creative Commons Attribution 4.0 International Public License](#).



### Book citation

You can use the following BibTex entry to cite this book:

```
@book{pedro2024,  
  author = {Pedro Duarte Faria},  
  title = {Introduction to pyspark},  
  month = {January},  
  year = {2024},  
  address = {Belo Horizonte}  
}
```

### Corresponding author and maintainer

Pedro Duarte Faria

Contact: [pedropark99@gmail.com](mailto:pedropark99@gmail.com)

Personal website: <https://pedro-faria.netlify.app/>

---

<sup>4</sup><https://phoenixnap.com/kb/install-spark-on-ubuntu>.

# 1 Key concepts of python

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.

## 1.1 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<sup>1</sup>, PyCharm<sup>2</sup>, Anaconda<sup>3</sup> and RStudio<sup>4</sup>). Many of these programs are free to use, and, are easy to install.

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.

## 1.2 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:

---

<sup>1</sup><https://code.visualstudio.com/>

<sup>2</sup><https://www.jetbrains.com/pycharm/>

<sup>3</sup><https://www.anaconda.com/products/distribution>

<sup>4</sup><https://www.rstudio.com/>

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

It will be much easier to run this script, if you open your OS's terminal inside the folder where you save the `hello.py` file. After you opened the terminal inside the folder, just run the `python3 hello.py` command. As a result, python will execute `hello.py`, and, the text `Hello World!` should be printed to the terminal:

```
Terminal$ python3 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.

## 1.3 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 your 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 liters',
    'price': 2.52,
    'group': 'non-alcoholic drinks',
    'department': 'drinks'
}

print(product)
```

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

And many other things...

## 1.4 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, then, goes to the second expression, and runs it, then 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)
```



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`.

## 1.5 Packages (or libraries)

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<sup>5</sup>. If a python package is published in PyPI, then, you can easily install it through the pip tool.

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.

```
import pandas
```

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

---

<sup>5</sup><https://pypi.org/>

If your program produce this 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.

```
pip install pandas
```

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:

```
# Now that I installed the `pandas` package with `pip`
# this `import` statement works without any errors:
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

```

Just to be clear, you can import multiple functions from the package, by listing them. Or, if you prefer, you can import all components of the package (or module/sub-module) by using the star shortcut (\*):

```

# Import `search()`, `match()` and `compile()` functions:
from re import search, match, compile
# Import all functions from the `os` package
from os import *

```

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 many “modules”. In other words, the functions and classes of this python package are usually organized in “modules”.

As an example, the pyspark package is a fairly large package, that contains many classes and functions. Because of it, the package is organized in a number of modules, such as sql (to access Spark SQL), pandas (to access the Pandas API of Spark), ml (to access Spark MLlib).

To access the functions available in each one of these modules, you use the dot operator between the name of the package and the name of the module. For example, to import all components from the sql and pandas modules of pyspark, you would do this:

```
from pyspark.sql import *
from pyspark.pandas import *
```

Going further, we can have sub-modules (or modules inside a module) too. As an example, the sql module of pyspark have the functions and window sub-modules. To access these sub-modules, you use the dot operator again:

```
# Importing `functions` and `window` sub-modules:
import pyspark.sql.functions as F
import pyspark.sql.window as W
```

## 1.6 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. This means that, in order 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 of 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

Note in the example above, that we access any class method in the same way that we would access a sub-module/module of a package. That is, by using the dot operator (`.`).

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'

## 1.7 Identifying classes and their methods

Over the next chapters, you will realize that pyspark programs tend to use more methods than standard functions. So most of the functionality of pyspark 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

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.

### 2.1 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<sup>1</sup> for performing large-scale data processing; a complete library for scalable machine learning (MLib<sup>2</sup>); a stream processing engine<sup>3</sup> 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 is often necessary in production data analysis pipelines. In addition, it reduces the management burden of maintaining separate tools (Karau et al. 2015).

---

<sup>1</sup><https://spark.apache.org/sql/>

<sup>2</sup><https://spark.apache.org/docs/latest/ml-guide.html>

<sup>3</sup><https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#overview>

## 2.2 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 let's 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 allows 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 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.



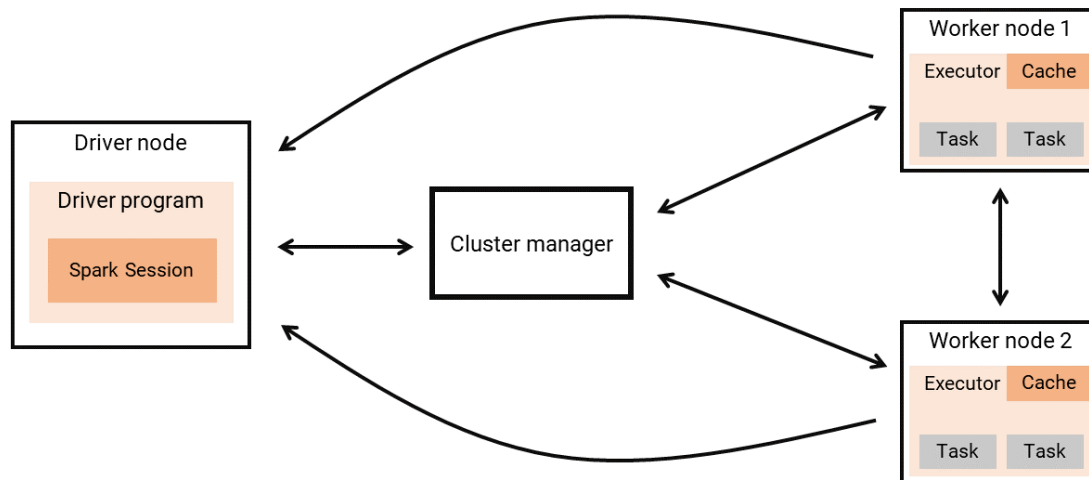


Figure 2.1: Spark application structure on a cluster of computers

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.

## 2.3 Spark application versus pyspark application

The pyspark package is just a tool to write Spark applications using the python programming language. This means, that every pyspark application is a Spark application written in python.

With this conception in mind, you can understand that a pyspark application is a description of a Spark application. When we compile (or execute) our python program, this description is translated into a raw Spark application that will be executed by Spark.

To write a pyspark application, you write a python script that uses the pyspark library. When you execute this python script with the python interpreter, the application will be automatically converted to Spark code, and will be sent to Spark to be executed across the cluster;

## 2.4 Core parts of a pyspark program

In this section, I want to point out the core parts that composes every pyspark program. This means that every pyspark program that you write will have these “core parts”, which are:

- 1) importing the pyspark package (or modules);
- 2) starting your Spark Session;
- 3) defining a set of transformations and actions over Spark DataFrames;

### 2.4.1 Importing the pyspark package (or modules)

Spark comes with a lot of functionality installed. But, in order to use it in your pyspark program, you have to import most of these functionalities to your session. This means that you have to import specific packages (or “modules”) of pyspark to your python session.

For example, most of the functions used to define our transformations and aggregations in Spark DataFrames, comes from the `pyspark.sql.functions` module.

That is why we usually start our python scripts by importing functions from this module, like this:

```
from pyspark.sql.functions import sum, col
sum_expr = sum(col('Value'))
```

Or, importing the entire module with the `import` keyword, like this:

```
import pyspark.sql.functions as F
sum_expr = F.sum(F.col('Value'))
```

### 2.4.2 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.Session.builder` module.

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()
```

### 2.4.3 Defining a set of transformations and actions

Every pyspark program is composed by a set of transformations and actions over a set of Spark DataFrames.

I will explain Spark DataFrames in more depth on the Chapter 3. For now just understand that they are the basic data structure that feed all pyspark programs. In other words, on every pyspark program we are transforming multiple Spark DataFrames to get the result we want.

As an example, in the script below we begin with the Spark DataFrame stored in the object `students`, and, apply multiple transformations over it to build the `ar_department` DataFrame. Lastly, we apply the `.show()` action over the `ar_department` DataFrame:

```
from pyspark.sql.functions import col
# Apply some transformations over
# the `students` DataFrame:
ar_department = students\
    .filter(col('Age') > 22)\
    .withColumn('IsArDepartment', col('Department') == 'AR')\
    .orderBy(col('Age').desc())

# Apply the `.show()` action
# over the `ar_department` DataFrame:
ar_department.show()
```

## 2.5 Building your first Spark application

To demonstrate what a pyspark program looks like, let's write and run our first example of a Spark application. This Spark application will build a simple table of 1 column that contains 5 numbers, and then, it will return a simple python list containing these five numbers as a result.

### 2.5.1 Writing the code

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 our first Spark table we use the `range()` method from the `spark` object. The `range()` method works similarly as the standard python function called `range()`. It basically creates a sequence of numbers, from 0 to  $n - 1$ . However, this `range()` method from `spark` stores this sequence of numbers as rows in a Spark table (or a Spark DataFrame):

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

After this step, we want to collect all the rows of the resulting table into a python list. And to do that, we use the `collect()` method from the Spark table:

```
result = table.collect()
print(result)
```

So, the entire program is composed of these three parts (or sections) of code. If you need it, the entire program is reproduced below. You can copy and paste all of this code to your python script, and then, save it:

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

table = spark.range(5)
result = table.collect()
print(result)
```

## 2.5.2 Executing the code

Now that you have written your first Spark application with `pyspark`, you want to execute this application and see its results. Yet, to run a `pyspark` program, remember that you need to have the necessary software installed on your machine. In case you do not have Apache Spark installed yet, I personally recommend you to read the [articles from PhoenixNAP on how to install Apache Spark](https://phoenixnap.com/kb/install-spark-on-ubuntu)<sup>4</sup>.

Anyway, to execute this `pyspark` that you wrote, you need send this script to the python interpreter, and to do this you need to: 1) open a terminal inside the folder where your python script is stored; and, 2) use the `python` command from the terminal with the name of your python script.

---

<sup>4</sup><https://phoenixnap.com/kb/install-spark-on-ubuntu>.

In my current situation, I'm running Spark on a Ubuntu distribution, and, I saved the `spark-example.py` script inside a folder called `SparkExample`. This folder is located at the path `~/Documentos/Projetos/Livros/Introd-pys` of my computer. This means that, I need to open a terminal that is rooted inside this `SparkExample` folder.

You probably have saved your `spark-example.py` file in a different folder of your computer. This means that you need to open the terminal from a different folder.

After I opened a terminal rooted inside the `SparkExample` folder. I just use the `python3` command to access the python interpreter, and, give the name of the python script that I want to execute. In this case, the `spark-example.py` file. As a result, our first pyspark program will be executed:

```
Terminal$ cd ../../SparkExample
Terminal$ python3 spark-example.py
```

```
[Row(id=0), Row(id=1), Row(id=2), Row(id=3), Row(id=4)]
```

You can see in the above result, that this Spark application produces a sequence of `Row` objects, inside a Python list. Each row object contains a number from 0 to 4.

Congratulations! You have just run your first Spark application using `pyspark`!

## 2.6 Overview of pyspark

Before we continue, I want to give you a very brief overview of the main parts of `pyspark` that are the most useful and most important to know of.

### 2.6.1 Main python modules

The main python modules that exist in `pyspark` are:

- `pyspark.sql.Session`: the `Session` class that defines your Spark Session, or, the entry point to your Spark application;
- `pyspark.sql.dataframe`: module that defines the `DataFrame` class;
- `pyspark.sql.column`: module that defines the `Column` class;
- `pyspark.sql.types`: module that contains all data types of Spark;
- `pyspark.sql.functions`: module that contains all of the main Spark functions that we use in transformations;
- `pyspark.sql.window`: module that defines the `Window` class, which is responsible for defining windows in a Spark `DataFrame`;

## 2.6.2 Main python classes

The main python classes that exists in pyspark are:

- `DataFrame`: represents a Spark `DataFrame`, and it is the main data structure in pyspark. In essence, they represent a collection of datasets into named columns;
- `Column`: represents a column in a Spark `DataFrame`;
- `GroupedData`: represents a grouped Spark `DataFrame` (result of `DataFrame.groupby()`);
- `Window`: describes a window in a Spark `DataFrame`;
- `DataFrameReader` and `DataFrameWriter`: classes responsible for reading data from a data source into a Spark `DataFrame`, and writing data from a Spark `DataFrame` into a data source;
- `DataFrameNaFunctions`: class that stores all main methods for dealing with null values (i.e. missing data);

## 3 Introducing Spark DataFrames

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, they 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, because much of your Spark applications will heavily use this API to compose your data transformations and data flows (Chambers and Zaharia 2018).

### 3.1 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, or, in spreadsheets (like Excel). 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 through 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, let's 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.

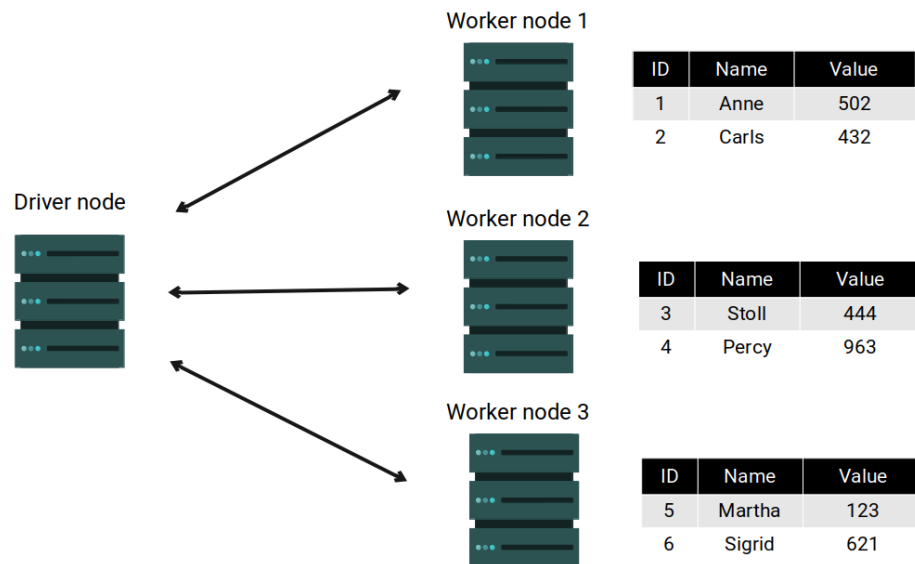


Figure 3.1: A Spark DataFrame is distributed across the cluster

## 3.2 Partitions of a Spark DataFrame

A Spark DataFrame is always broken into many small pieces, and, these pieces are always spread across the cluster of machines. Each one of these small pieces of the total data are considered a DataFrame *partition*.



For the most part, you do not manipulate these partitions manually or individually (Karau et al. 2015), because Spark automatically do this job for you.

As we exposed in Figure 3.1, each node of the cluster will hold a piece of the total DataFrame. If we translate this distribution into a “partition” distribution, this means that each node of the cluster can hold one or multiple partitions of the Spark DataFrame.

If we sum all partitions present in a node of the cluster, we get a chunk of the total DataFrame. The figure below demonstrates this notion:

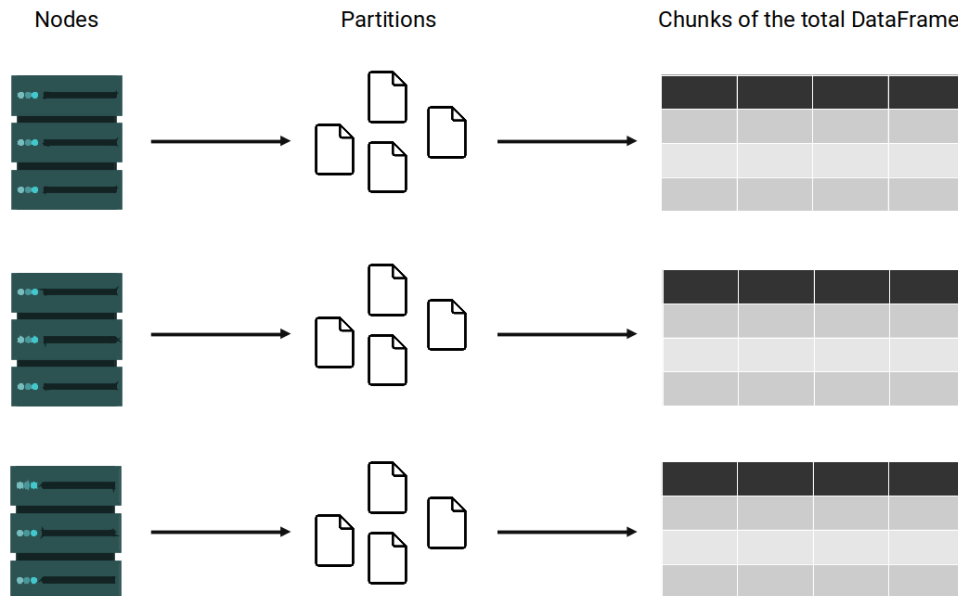


Figure 3.2: Partitions of a DataFrame

If the Spark DataFrame is not big, each node of the cluster will probably store just a single partition of this DataFrame. In contrast, depending on the complexity and size of the DataFrame, Spark will split this DataFrame into more partitions than there are nodes in the cluster. In this case, each node of the cluster will hold more than 1 partition of the total DataFrame.

### 3.3 The DataFrame class in pyspark

In pyspark, every Spark DataFrame is stored inside a python object of class `pyspark.sql.dataframe.DataFrame`. Or more succinctly, a object of class `DataFrame`.

Like any python class, the DataFrame class comes with multiple methods that are available for every object of this class. This means that you can use any of these methods in any Spark DataFrame that you create through pyspark.

As an example, in the code below I expose all the available methods from this DataFrame class. First, I create a Spark DataFrame with `spark.range(5)`, and, store it in the object `df5`. After that, I use the `dir()` function to show all the methods that I can use through this `df5` object:

```
df5 = spark.range(5)
available_methods = dir(df5)
print(available_methods)
```

```
['__class__', '__delattr__', '__dict__',
 '__dir__', '__doc__', '__eq__',
 '__format__', '__ge__', '__getattr__',
 '__getattribute__', '__getitem__', '__gt__',
 '__hash__', '__init__', '__init_subclass__',
 '__le__', '__lt__', '__module__',
 '__ne__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__', '__setattr__',
 '__sizeof__', '__str__', '__subclasshook__',
 '__weakref__', '_collect_as_arrow', '_ipython_key_completions_',
 '_jcols', '_jdf', '_jmap',
 '_joinAsOf', '_jseq', '_lazy_rdd',
 '_repr_html_', '_sc', '_schema',
 '_session', '_sort_cols', '_sql_ctx',
 '_support_repr_html', 'agg', 'alias',
 'approxQuantile', 'cache', 'checkpoint',
 'coalesce', 'colRegex', 'collect',
 'columns', 'corr', 'count',
 'cov', 'createGlobalTempView', 'createOrReplaceGlobalTempView',
 'createOrReplaceTempView', 'createTempView', 'crossJoin',
 'crosstab', 'cube', 'describe',
 'distinct', 'drop', 'dropDuplicates',
 'dropDuplicatesWithinWatermark', 'drop_duplicates', 'dropna',
 'dtypes', 'exceptAll', 'explain',
 'fillna', 'filter', 'first',
 'foreach', 'foreachPartition', 'freqItems',
 'groupBy', 'groupby', 'head',
 'hint', 'id', 'inputFiles',
 'intersect', 'intersectAll', 'isEmpty',
 'isLocal', 'isStreaming', 'is_cached',
```

```

'join', 'limit', 'localCheckpoint',
'mapInArrow', 'mapInPandas', 'melt',
'na', 'observe', 'offset',
'orderBy', 'pandas_api', 'persist',
'printSchema', 'randomSplit', 'rdd',
'registerTempTable', 'repartition', 'repartitionByRange',
'replace', 'rollup', 'sameSemantics',
'sample', 'sampleBy', 'schema',
'select', 'selectExpr', 'semanticHash',
'show', 'sort', 'sortWithinPartitions',
'sparkSession', 'sql_ctx', 'stat',
'storageLevel', 'subtract', 'summary',
'tail', 'take', 'to',
'toDF', 'toJSON', 'toLocalIterator',
'toPandas', 'to_koalas', 'to_pandas_on_spark',
'transform', 'union', 'unionAll',
'unionByName', 'unpersist', 'unpivot',
'where', 'withColumn', 'withColumnRenamed',
'withColumns', 'withColumnsRenamed', 'withMetadata',
'withWatermark', 'write', 'writeStream',
'writeTo']

```

All the methods present in this `DataFrame` class, are commonly referred as the *DataFrame API of Spark*. Remember, this is the most important API of Spark. Because much of your Spark applications will heavily use this API to compose your data transformations and data flows (Chambers and Zaharia 2018).

### 3.4 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 Rows, 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)

```

Remember that a Spark DataFrame in python is a object of class `pyspark.sql.dataframe.DataFrame` as you can see below:

```
type(df)
```

```
pyspark.sql.dataframe.DataFrame
```

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:

```
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` from two different python lists (`data` and `columns`).

The first list (`data`) is a list of rows. Each row is represent by a python tuple, which contains the values in each column. But the secont list (`columns`) contains the names for each column in the DataFrame.

To create the `students` DataFrame we deliver these two lists to `createDataFrame()` method:

```

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, Score2: bigint, Score3: bigint, Score4: bigint, Course: string, Departme  
nt: string]
```

You can also use a method that returns a `DataFrame` object by default. Examples are the `table()` and `range()` methods from your Spark Session, like we used in the [Section 3.3](#), to create the `df5` object.

Other examples are the methods used to read data and import it to pyspark. These methods are available in the `spark.read` module, like `spark.read.csv()` and `spark.read.json()`. These methods will be described in more depth in [Chapter 6](#).

## 3.5 Visualizing 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 explicitly ask for it with a trigger operation, which is called an “action” (this kind of operation is described in [Section 5.2](#)).

You can notice this laziness in the output below:

```
students
```

```
DataFrame[StudentID: bigint, Name: string, Age: bigint, Height: double, Score1:  
bigint, Score2: bigint, Score3: bigint, Score4: bigint, Course: string, Departme  
nt: string]
```

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? For this, 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()
```

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

## 3.6 Getting the name of the columns

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']
```

## 3.7 Getting the number of rows

If you want to know the number of rows present in a Spark DataFrame, just use the `count()` method of this DataFrame. As a result, Spark will build this DataFrame, and count the number of rows present in it.

```
students.count()
```

6

## 3.8 Spark Data Types

Each column of your Spark DataFrame is associated with a specific data type. Spark supports a large number of different data types. You can see the full list at the official documentation page<sup>1</sup>. 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 that it can represent is from -2147483648 to 2147483647.
- `LongType`: Represents 8-byte signed integer numbers. The range of numbers that it can represent 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.

---

<sup>1</sup>The full list is available at the link <https://spark.apache.org/docs/3.3.0/sql-ref-datatypes.html#supported-data-types>

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 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 `StringType`, we can do this:

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

`StringType()`

## 3.9 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, they can be automatically defined by Spark while reading or creating your 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. Furthermore, by the descriptions `value: double` and `date: date`, we know too that the columns `value` and `date` are of type `DoubleType()` and `DateType()`, respectively.

```
df
```

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

You can also visualize a more complete report of the `DataFrame` schema by using the `printSchema()` method, like this:



```
df.printSchema()
```

```
root
|-- id: long (nullable = true)
|-- value: double (nullable = true)
|-- date: date (nullable = true)
```

### 3.9.1 Accessing the DataFrame schema

So, by calling the object of your DataFrame (i.e. an object of class DataFrame) you can see a small description of the schema of this DataFrame. But, how can you access this schema programmatically?

You do this, by using the schema method of your DataFrame, like in the example below:

```
df.schema
```

```
StructType([StructField('id', LongType(), True),
StructField('value', DoubleType(), True),
StructField('date', DateType(), True)])
```

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 stores 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 about 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
```

### 3.9.2 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()` constructors, 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.

You can see below that I deliver this schema object that I built to `spark.createDataFrame()`. Now `spark.createDataFrame()` will follow the schema I described in this schema object when building the registers DataFrame.

```
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)
```

Having this example in mind, in order to build a DataFrame schema from scratch, you have to build the equivalent `StructType()` object that represents the schema you want.

### 3.9.3 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 if 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 data type  $x$ , 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  $x$ . Lets use 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. But, you can see in the result below, that 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

This unexpected result happens, because the id column is actually from the “big integer” type, or, the LongType (which are 8-byte signed integer). You can see below, that now the test results in true:

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

True

## 4 Introducing the Column class

As we described at the introduction of Chapter 3, you will massively use the methods from the `DataFrame` class in your Spark applications to manage, modify and calculate your Spark DataFrames.

However, 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 your Spark `DataFrame` is a object of class `Column`.

We can confirm this statement, by taking the `df` `DataFrame` that we showed at Section 3.4, and look at the class of any column of it. Like the `id` column:

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

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

### 4.1 Building a column object

You can refer to or create a column, by using the `col()` and `column()` functions from `pyspark.sql.functions` module. These functions receive a string input with the name of the column you want to create/refer to.

Their result are always a object of class `Column`. For example, the code below creates a column called `ID`:

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

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

## 4.2 Columns are strongly related to expressions

Many kinds of transformations that we want to apply over a Spark DataFrame, are usually described through expressions, and, these expressions in Spark are mainly composed by **column transformations**. That is why the Column class, and its methods, are so important in Apache Spark.

Columns in Spark are so strongly related to expressions that the columns themselves are initially interpreted as expressions. If we look again at the column `id` from `df` DataFrame, Spark will bring an expression as a result, and not the values hold by this column.

```
df.id
```

```
Column<'id'>
```

Having these ideas in mind, when I created the column `ID` on the previous section, I created a “column expression”. This means that `col("ID")` is just an expression, and as consequence, Spark does not know which are the values of column `ID`, or, where it lives (which is the DataFrame that this column belongs?). For now, Spark is not interested in this information, it just knows that we have an expression referring to a column called `ID`.

These ideas relates a lot to the **lazy aspect** of Spark that we talked about in Section 3.5. Spark will not perform any heavy calculation, or show you the actual results/values from your code, until you trigger the calculations with an action (we will talk more about these “actions” on Section 5.2). As a result, when you access a column, Spark will only deliver an expression that represents that column, and not the actual values of that column.

This is handy, because we can store our expressions in variables, and, reuse it latter, in multiple parts of our code. For example, I can keep building and merging a column with different kinds of operators, to build a more complex expression. In the example below, I create an expression that doubles the values of `ID` column:

```
expr1 = id_column * 2
print(expr1)
```

```
Column<'(ID * 2)'>
```

Remember, with this expression, Spark knows that we want to get a column called `ID` somewhere, and double its values. But Spark will not perform that action right now.

Logical expressions follow the same logic. In the example below, I am looking for rows where the value in column `Name` is equal to 'Anne', and, the value in column `Grade` is above 6.

Again, Spark just checks if this is a valid logical expression. For now, Spark does not want to know where are these Name and Grade columns. Spark does not evaluate the expression, until we ask for it with an action:

```
expr2 = (col('Name') == 'Anne') & (col('Grade') > 6)
print(expr2)
```

```
Column<'((Name = Anne) AND (Grade > 6))'>
```

## 4.3 Literal values versus expressions

We know now that columns of a Spark DataFrame have a deep connection with expressions. But, on the other hand, there are some situations that you write a value (it can be a string, a integer, a boolean, or anything) inside your pyspark code, and you might actually want Spark to interpret this value as a constant (or a literal) value, rather than a expression.

As an example, lets suppose you control the data generated by the sales of five different stores, scattered across different regions of Belo Horizonte city (in Brazil). Now, lets suppose you receive a batch of data generated by the 4th store in the city, which is located at Amazonas Avenue, 324. This batch of data is exposed below:

```
path = '../Data/sales.json'
sales = spark.read.json(path)
sales.show(5)
```

```
+-----+-----+-----+-----+-----+
|price|product_id|product_name|sale_id|          timestamp|units|
+-----+-----+-----+-----+-----+
| 3.12|      134| Milk 1L Mua| 328711|2022-02-01T22:10:02|   1|
| 1.22|      110|  Coke 350ml| 328712|2022-02-03T11:42:09|   3|
| 4.65|      117|   Pepsi 2L| 328713|2022-02-03T14:22:15|   1|
| 1.22|      110|  Coke 350ml| 328714|2022-02-03T18:33:08|   1|
| 0.85|      341|Trident Mint| 328715|2022-02-04T15:41:36|   1|
+-----+-----+-----+-----+-----+
```

If you look at this batch... there is no indication that these sales come from the 4th store. In other words, this information is not present in the data, is just in your mind. It certainly is a very bad idea to leave this data as is, without any identification of the source of it. So, you might want to add some labels and new columns to this batch of data, that can easily identify the store that originated these sales.

For example, we could add two new columns to this sales DataFrame. One for the number that identifies the store (4), and, another to keep the store address. Considering that all rows in this batch comes from the 4th store, we should add two “constant” columns, meaning that these columns should have a constant value across all rows in this batch. But, how can we do this? How can we create a “constant” column? The answer is: by forcing Spark to interpret the values as literal values, instead of a expression.

In other words, I can not use the `col()` function to create these two new columns. Because this `col()` function receives a column name as input. **It interprets our input as an expression that refers to a column name.** This function does not accept some sort of description of the actual values that this column should store.

## 4.4 Passing a literal (or a constant) value to Spark

So how do we force Spark to interpret a value as a literal (or constant) value, rather than a expression? To do this, you must write this value inside the `lit()` (short for “literal”) function from the `pyspark.sql.functions` module.

In other words, when you write in your code the statement `lit(4)`, Spark understand that you want to create a new column which is filled with 4’s. In other words, this new column is filled with the constant integer 4.

With the code below, I am creating two new columns (called `store_number` and `store_address`), and adding them to the sales DataFrame.

```
from pyspark.sql.functions import lit
store_number = lit(4).alias('store_number')
store_address = lit('Amazonas Avenue, 324').alias('store_address')

sales = sales\
    .select(
        '*', store_number, store_address
    )

sales\
    .select(
        'product_id', 'product_name',
        'store_number', 'store_address'
    )\
    .show(5)
```

+-----+-----+-----+-----+



product_id	product_name	store_number	store_address
134	Milk 1L Mua	4	Amazonas Avenue, 324
110	Coke 350ml	4	Amazonas Avenue, 324
117	Pepsi 2L	4	Amazonas Avenue, 324
110	Coke 350ml	4	Amazonas Avenue, 324
341	Trident Mint	4	Amazonas Avenue, 324

In essence, you normally use the `lit()` function when you want to write a literal value in places where Spark expects a column name. In the example above, instead of writing a name to an existing column in the sales DataFrame, I wanted to write the literal values 'Amazonas Avenue, 324' and 4, and I used the `lit()` function to make this intention very clear to Spark. If I did not use the `lit()` function, the `withColumn()` method would interpret the value 'Amazonas Avenue, 324' as an existing column named Amazonas Avenue, 324.

## 4.5 Key methods of the Column class

Because many transformations that we want to apply over our DataFrames are expressed as column transformations, the methods from the Column class will be quite useful on many different contexts. You will see many of these methods across the next chapters, like `desc()`, `alias()` and `cast()`.

Remember, you can always use the `dir()` function to see the complete list of methods available in any python class. It is always useful to check the official documentation too<sup>1</sup>. There you will have a more complete description of each method.

But since they are so important in Spark, let's just give you a brief overview of some of the most popular methods from the Column class (these methods will be described in more detail in later chapters):

- `desc()` and `asc()`: methods to order the values of the column in a descending or ascending order (respectively);
- `cast()` and `astype()`: methods to cast (or convert) the values of the column to a specific data type;
- `alias()`: method to rename a column;
- `substr()`: method that returns a new column with the sub string of each value;
- `isNull()` and `isNotNull()`: logical methods to test if each value in the column is a null value or not;
- `startswith()` and `endswith()`: logical methods to search for values that start with or end with a specific pattern;
- `like()` and `rlike()`: logical methods to search for a specific pattern or regular expression in the values of the column;

<sup>1</sup><https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Column.html>

- `isin()`: logical method to test if each value in the column is some of the listed values;

# 5 Transforming your Spark DataFrame - Part 1

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.

This means that you 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.

## 5.1 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 six basic transformations that you can apply to a DataFrame:

- Filtering rows based on a logical condition;
- Selecting a subset of rows;
- Selecting specific columns;
- Adding or deleting columns;
- Sorting rows;
- Calculating 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)

```

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()`, `orderBy()` and many others. Remember, these are methods from the python class that defines Spark DataFrame's (i.e. the `pyspark.sql.dataframe.DataFrame` class).

This means that you can apply these transformations only to Spark DataFrames, and no other kind of python object. For example, if you try to use the `orderBy()` method in a standard python string (i.e. an object of class `str`), you will get an `AttributeError` error. Because this class of object in python, does not have a `orderBy()` method:

```

s = "A python string"
s.orderBy('value')

```

Traceback (most recent call last):

```

File "<stdin>", line 1, in <module>
AttributeError: 'str' object has no attribute 'orderBy'

```

Each one of these DataFrame 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 until you trigger these transformations with an **action**.

## 5.2 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()
```

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

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()
```

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=20.1), Row(date=datetime.date(2021,
1, 1), value=28.3)]
```

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 of a Spark DataFrame, but we will explain this method latter at [Chapter 6](#).

## 5.3 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, or not.

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.

## Narrow transformations

are 1 to 1 transformations

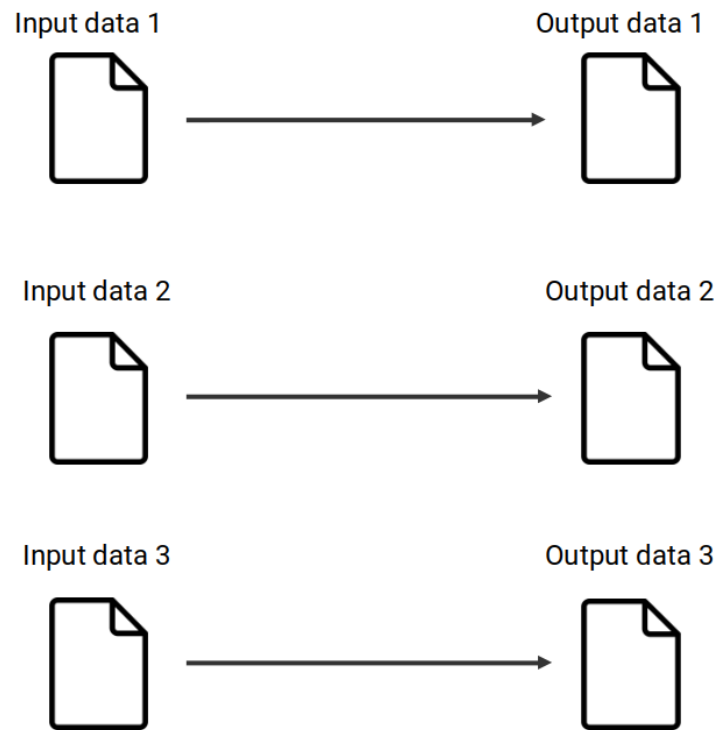


Figure 5.1: Presenting narrow transformations

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.

## 5.4 The transf DataFrame

To demonstrate some of the next examples in this chapter, we will use a different DataFrame called `transf`. The data that represents this DataFrame is freely available as a CSV file. You can download this CSV at the repository of this book<sup>1</sup>.

---

<sup>1</sup><https://github.com/pedropark99/Introd-pyspark/tree/main/Data>



## Wide transformations

are 1 to N transformations

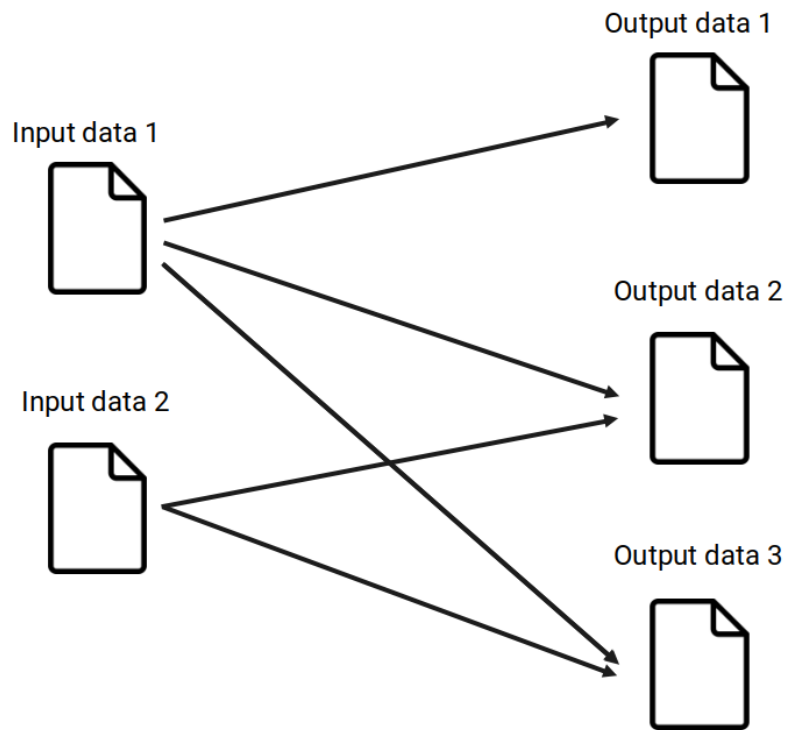


Figure 5.2: Presenting wide transformations

With the code below, you can import the data from the `transf.csv` CSV file, to recreate the `transf` DataFrame in your Spark Session:

```
from pyspark.sql.types import StructType, StructField
from pyspark.sql.types import DoubleType, StringType
from pyspark.sql.types import LongType, TimestampType, DateType
path = "../Data/transf.csv"
schema = StructType([
    StructField('dateTransfer', DateType(), False),
    StructField('datetimeTransfer', TimestampType(), False),
    StructField('clientNumber', LongType(), False),
    StructField('transferValue', DoubleType(), False),
    StructField('transferCurrency', StringType(), False),
    StructField('transferID', LongType(), False),
    StructField('transferLog', StringType(), False),
    StructField('destinationBankNumber', LongType(), False),
    StructField('destinationBankBranch', LongType(), False),
    StructField('destinationBankAccount', StringType(), False)
])

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

You could also use the `pandas` library to read the DataFrame directly from GitHub, without having to manually download the file:

```
import pandas as pd
url = 'https://raw.githubusercontent.com/'
url = url + 'pedropark99/Introd-pyspark/'
url = url + 'main/Data/transf.csv'

transf_pd = pd.read_csv(
    url, sep = ';',
    dtype = str,
    keep_default_na = False
)

transf_pd['transferValue'] = transf_pd['transferValue'].astype('float')

columns_to_int = [
    'clientNumber',
    'transferID',
```

```

        'destinationBankNumber',
        'destinationBankBranch'
    ]
    for column in columns_to_int:
        transf_pd[column] = transf_pd[column].astype('int')

    from pyspark.sql import SparkSession
    from pyspark.sql.functions import col
    spark = SparkSession.builder.getOrCreate()
    transf = spark.createDataFrame(transf_pd)\
        .withColumn('dateTransfer', col('dateTransfer').cast('date'))\
        .withColumn('datetimeTransfer', col('datetimeTransfer').cast('timestamp'))

```

This transf DataFrame contains bank transfer records from a fictitious bank. Before I show you the actual data of this DataFrame, is useful to give you a quick description of each column that it contains:

- dateTransfer: the date when the transfer occurred;
- datetimeTransfer: the date and time when the transfer occurred;
- clientNumber the unique number that identifies a client of the bank;
- transferValue: the nominal value that was transferred;
- transferCurrency: the currency of the nominal value transferred;
- transferID: an unique ID for the transfer;
- transferLog: store any error message that may have appeared during the execution of the transfer;
- destinationBankNumber: the transfer destination bank number;
- destinationBankBranch: the transfer destination branch number;
- destinationBankAccount: the transfer destination account number;

Now, to see the actual data of this DataFrame, we can use the show() action as usual.

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

```

+-----+-----+-----+-----+-----+
|dateTransfer|  datetimeTransfer|clientNumber|transferValue|transferCurrency|
+-----+-----+-----+-----+-----+
|  2022-12-31|2022-12-31 14:00:24|      5516|      7794.31|          zing f|
|  2022-12-31|2022-12-31 10:32:07|      4965|      7919.0|          zing f|
|  2022-12-31|2022-12-31 07:37:02|      4608|      5603.0|        dollar $|
|  2022-12-31|2022-12-31 07:35:05|      1121|      4365.22|        dollar $|
|  2022-12-31|2022-12-31 02:53:44|      1121|      4620.0|        dollar $|
+-----+-----+-----+-----+-----+

```

only showing top 5 rows

```
... with 5 more columns: transferID, transferLog, destinationBankNumber
    destinationBankBranch, destinationBankAccount
```

As you can see below, this transf DataFrame have 2421 rows in total:

```
transf.count()
```

2421

## 5.5 Filtering rows of your DataFrame

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

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

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

```
+-----+-----+-----+-----+-----+
|dateTransfer|  datetimeTransfer|clientNumber|transferValue|transferCurrency|
+-----+-----+-----+-----+-----+
|  2022-12-18|2022-12-18 08:45:30|      1297|      142.66|      dollar $|
|  2022-12-13|2022-12-13 20:44:23|      5516|      992.15|      dollar $|
|  2022-11-24|2022-11-24 20:01:39|      1945|      174.64|      dollar $|
|  2022-11-07|2022-11-07 16:35:57|      4862|      570.69|      dollar $|
|  2022-11-04|2022-11-04 20:00:34|      1297|       854.0|      dollar $|
+-----+-----+-----+-----+-----+
```

only showing top 5 rows

```
... with 5 more columns: transferID, transferLog, destinationBankNumber
    destinationBankBranch, destinationBankAccount
```

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 also write the same exact expression in a more “pythonic” way, using the `col()` function from `pyspark.sql.functions` module.

```
from pyspark.sql.functions import col

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

```
+-----+-----+-----+-----+-----+
|dateTransfer|    datetimeTransfer|clientNumber|transferValue|transferCurrency|
+-----+-----+-----+-----+-----+
| 2022-12-18|2022-12-18 08:45:30|      1297|      142.66|      dollar $|
| 2022-12-13|2022-12-13 20:44:23|      5516|      992.15|      dollar $|
| 2022-11-24|2022-11-24 20:01:39|      1945|      174.64|      dollar $|
| 2022-11-07|2022-11-07 16:35:57|      4862|      570.69|      dollar $|
| 2022-11-04|2022-11-04 20:00:34|      1297|       854.0|      dollar $|
+-----+-----+-----+-----+-----+
```

only showing top 5 rows

... with 5 more columns: transferID, transferLog, destinationBankNumber  
destinationBankBranch, destinationBankAccount

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.transferValue < 1000)
```

### 5.5.1 Logical operators available

As we saw in the previous section, there are two ways to write logical expressions in pyspark: 1) write a SQL logical expression inside a string; 2) or, write a python logical expression using the `col()` function.

If you choose to write a 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 instead.

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

Table 5.1: 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 than or equal to y?
>=	x >= y	is x greater than or equal to 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?

And, the logical operators of python are described in the table below:

Table 5.2: 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 than or equal to y?
>=	x >= y	is x greater than or equal to 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?

### 5.5.2 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 from client of number 1297 where the transfer value is smaller than 1000, and the date of the transfer is after 20 of February 2022. These conditions are dependent, that is, they are connected to each other (the client number, the transfer value and the date of the transfer). That is why I used the `and` keyword between each condition in the example below (i.e. to connect these three conditions together).

```

condition = '''
    transferValue < 1000
    and clientNumber == 1297
    and dateTransfer > '2022-02-20'
'''

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

```

```

+-----+-----+-----+-----+-----+
|dateTransfer|  datetimeTransfer|clientNumber|transferValue|transferCurrency|
+-----+-----+-----+-----+-----+
|  2022-12-18|2022-12-18 08:45:30|      1297|      142.66|      dollar $|
|  2022-11-04|2022-11-04 20:00:34|      1297|       854.0|      dollar $|
|  2022-02-27|2022-02-27 13:27:44|      1297|       697.21|      dollar $|
+-----+-----+-----+-----+-----+
... with 5 more columns: transferID, transferLog, destinationBankNumber
                        destinationBankBranch, destinationBankAccount

```

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('transferValue') < 1000) &
        (col('clientNumber') == 1297) &
        (col('dateTransfer') > '2022-02-20')
    )\
    .show()

```

```

+-----+-----+-----+-----+-----+
|dateTransfer|  datetimeTransfer|clientNumber|transferValue|transferCurrency|
+-----+-----+-----+-----+-----+
|  2022-12-18|2022-12-18 08:45:30|      1297|      142.66|      dollar $|
|  2022-11-04|2022-11-04 20:00:34|      1297|       854.0|      dollar $|
|  2022-02-27|2022-02-27 13:27:44|      1297|       697.21|      dollar $|
+-----+-----+-----+-----+-----+
... with 5 more columns: transferID, transferLog, destinationBankNumber
                        destinationBankBranch, destinationBankAccount

```

This a **very important detail**, because it is very easy to forget. When building your complex logical expressions in the “python” way, 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('transferValue') < 1000 &  
    col('clientNumber') == 1297 &  
    col('dateTransfer') > '2022-02-20'  
  )\  
  .show(5)
```

Py4JError: An error occurred while calling o216.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:1  
82)  
  at py4j.ClientServerConnection.run(ClientServerConnection.java:106)  
  at java.base/java.lang.Thread.run(Thread.java:829)
```

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 transferValue is smaller than 1000, or, clientNumber is equal to 1297, or, dateTransfer is greater than 20 of February 2022.

```
condition = '''  
  transferValue < 1000  
  or clientNumber == 1297  
  or dateTransfer > '2022-02-20'  
  '''  
  
transf\  
  .filter(condition)\  
  .show(5)
```

```
+-----+-----+-----+-----+-----+  
|dateTransfer|  datetimeTransfer|clientNumber|transferValue|transferCurrency|
```



dateTransfer	datetimeTransfer	clientNumber	transferValue	transferCurrency
2022-12-31	2022-12-31 14:00:24	5516	7794.31	zing f
2022-12-31	2022-12-31 10:32:07	4965	7919.0	zing f
2022-12-31	2022-12-31 07:37:02	4608	5603.0	dollar \$
2022-12-31	2022-12-31 07:35:05	1121	4365.22	dollar \$
2022-12-31	2022-12-31 02:53:44	1121	4620.0	dollar \$

only showing top 5 rows

... with 5 more columns: transferID, transferLog, destinationBankNumber  
destinationBankBranch, destinationBankAccount

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('transferValue') < 1000) |
    (col('clientNumber') == 1297) |
    (col('dateTransfer') > '2022-02-20')
  )\
  .show(5)
```

dateTransfer	datetimeTransfer	clientNumber	transferValue	transferCurrency
2022-12-31	2022-12-31 14:00:24	5516	7794.31	zing f
2022-12-31	2022-12-31 10:32:07	4965	7919.0	zing f
2022-12-31	2022-12-31 07:37:02	4608	5603.0	dollar \$
2022-12-31	2022-12-31 07:35:05	1121	4365.22	dollar \$
2022-12-31	2022-12-31 02:53:44	1121	4620.0	dollar \$

only showing top 5 rows

... with 5 more columns: transferID, transferLog, destinationBankNumber  
destinationBankBranch, destinationBankAccount

You can also increase the complexity of your logical expressions by mixing dependent expressions with independent expressions. For example, to filter all the rows where dateTransfer is greater than or equal to 01 of October 2022, and clientNumber is either 2727 or 5188, you would have the following code:

```

condition = '''
    (clientNumber == 2727 or clientNumber == 5188)
    and dateTransfer >= '2022-10-01'
'''

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

```

```

+-----+-----+-----+-----+-----+
|dateTransfer|  datetimeTransfer|clientNumber|transferValue|transferCurrency|
+-----+-----+-----+-----+-----+
|  2022-12-29|2022-12-29 10:22:02|      2727|      4666.25|          euro €|
|  2022-12-27|2022-12-27 03:58:25|      5188|      7821.69|          dollar $|
|  2022-12-26|2022-12-25 23:45:02|      2727|      3261.73| british pound £|
|  2022-12-23|2022-12-23 05:32:49|      2727|       8042.0|          dollar $|
|  2022-12-22|2022-12-22 06:02:47|      5188|      8175.67|          dollar $|
+-----+-----+-----+-----+-----+

```

only showing top 5 rows

... with 5 more columns: transferID, transferLog, destinationBankNumber  
 destinationBankBranch, destinationBankAccount

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 `clientNumber` is equal to one of the listed values (2727 or 5188), and, that `dateTransfer` is greater than or equal to 01 of October 2022.

```

condition = '''
    clientNumber in (2727, 5188)
    and dateTransfer >= '2022-10-01'
'''

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

```

```

+-----+-----+-----+-----+-----+
|dateTransfer|  datetimeTransfer|clientNumber|transferValue|transferCurrency|
+-----+-----+-----+-----+-----+
|  2022-12-29|2022-12-29 10:22:02|      2727|      4666.25|          euro €|
|  2022-12-27|2022-12-27 03:58:25|      5188|      7821.69|          dollar $|

```

2022-12-26	2022-12-25 23:45:02	2727	3261.73	british pound £
2022-12-23	2022-12-23 05:32:49	2727	8042.0	dollar \$
2022-12-22	2022-12-22 06:02:47	5188	8175.67	dollar \$

only showing top 5 rows

... with 5 more columns: transferID, transferLog, destinationBankNumber  
destinationBankBranch, destinationBankAccount

### 5.5.3 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 `destinationBankNumber` is 290 or 666:

```
transf\
  .filter(col('destinationBankNumber').isin(290, 666))\
  .show(5)
```

dateTransfer	datetimeTransfer	clientNumber	transferValue	transferCurrency
2022-12-31	2022-12-31 07:37:02	4608	5603.0	dollar \$
2022-12-31	2022-12-31 07:35:05	1121	4365.22	dollar \$
2022-12-31	2022-12-31 02:44:46	1121	7158.0	zing f
2022-12-31	2022-12-31 01:02:06	4862	6714.0	dollar \$
2022-12-31	2022-12-31 00:48:47	3294	10882.52	dollar \$

only showing top 5 rows

... with 5 more columns: transferID, transferLog, destinationBankNumber  
destinationBankBranch, destinationBankAccount

### 5.5.4 Negating logical conditions

In some cases, is easier to describe what rows you **do not want** in your filter. That is, 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 `clientNumber` is not equal to 3284. Remember, the methods `filter()` and `where()` are equivalents or synonymous (they both mean the same thing).

```
condition = '''
    not clientNumber == 3284
'''

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

dateTransfer	datetimeTransfer	clientNumber	transferValue	transferCurrency
2022-12-31	2022-12-31 14:00:24	5516	7794.31	zing f
2022-12-31	2022-12-31 10:32:07	4965	7919.0	zing f
2022-12-31	2022-12-31 07:37:02	4608	5603.0	dollar \$
2022-12-31	2022-12-31 07:35:05	1121	4365.22	dollar \$
2022-12-31	2022-12-31 02:53:44	1121	4620.0	dollar \$

only showing top 5 rows

... with 5 more columns: `transferID`, `transferLog`, `destinationBankNumber`  
`destinationBankBranch`, `destinationBankAccount`

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

```
transf\
    .where(~(col('clientNumber') == 3284))\
    .show(5)
```

dateTransfer	datetimeTransfer	clientNumber	transferValue	transferCurrency
2022-12-31	2022-12-31 14:00:24	5516	7794.31	zing f
2022-12-31	2022-12-31 10:32:07	4965	7919.0	zing f
2022-12-31	2022-12-31 07:37:02	4608	5603.0	dollar \$
2022-12-31	2022-12-31 07:35:05	1121	4365.22	dollar \$
2022-12-31	2022-12-31 02:53:44	1121	4620.0	dollar \$

only showing top 5 rows

```
... with 5 more columns: transferID, transferLog, destinationBankNumber
    destinationBankBranch, destinationBankAccount
```

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

```
transf\
  .where(~col('clientNumber') == 3284)\
  .show(5)
```

```
AnalysisException: cannot resolve '(NOT clientNumber)' due to data type mismatch
: argument 1 requires boolean type, however, 'clientNumber' is of bigint type.;
'Filter (NOT clientNumber#210L = 3284)
```

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 'clientNumber' is not equal to
# 2727 or 5188.
transf\
  .where( (col('clientNumber').isin(2727, 5188)) == False )\
  .show(5)
```

```
+-----+-----+-----+-----+-----+
|dateTransfer|  datetimeTransfer|clientNumber|transferValue|transferCurrency|
+-----+-----+-----+-----+-----+
|  2022-12-31|2022-12-31 14:00:24|      5516|      7794.31|          zing f|
|  2022-12-31|2022-12-31 10:32:07|      4965|       7919.0|          zing f|
|  2022-12-31|2022-12-31 07:37:02|      4608|       5603.0|        dollar $|
|  2022-12-31|2022-12-31 07:35:05|      1121|      4365.22|        dollar $|
|  2022-12-31|2022-12-31 02:53:44|      1121|       4620.0|        dollar $|
+-----+-----+-----+-----+-----+
```

only showing top 5 rows

```
... with 5 more columns: transferID, transferLog, destinationBankNumber
    destinationBankBranch, destinationBankAccount
```

### 5.5.5 Filtering null values (i.e. missing data)

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('transferLog is null')\  
  .show(5)
```

dateTransfer	datetimeTransfer	clientNumber	transferValue	transferCurrency
2022-12-31	2022-12-31 14:00:24	5516	7794.31	zing f
2022-12-31	2022-12-31 10:32:07	4965	7919.0	zing f
2022-12-31	2022-12-31 07:37:02	4608	5603.0	dollar \$
2022-12-31	2022-12-31 07:35:05	1121	4365.22	dollar \$
2022-12-31	2022-12-31 02:53:44	1121	4620.0	dollar \$

only showing top 5 rows

... with 5 more columns: transferID, transferLog, destinationBankNumber  
destinationBankBranch, destinationBankAccount

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 transferLog is null')\  
  .show(5)
```

dateTransfer	datetimeTransfer	clientNumber	transferValue	transferCurrency
2022-12-05	2022-12-05 00:51:00	2197	8240.62	zing f
2022-09-20	2022-09-19 21:59:51	5188	7583.9	dollar \$
2022-09-03	2022-09-03 06:07:59	3795	3654.0	zing f
2022-07-02	2022-07-02 15:29:50	4465	5294.0	dollar \$

2022-06-14	2022-06-14 10:21:55	1121	7302.0	dollar \$
------------	---------------------	------	--------	-----------

only showing top 5 rows

... with 5 more columns: transferID, transferLog, destinationBankNumber  
destinationBankBranch, destinationBankAccount

The is and not keywords in SQL 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, the same expression above written in this form:

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

Both forms are equivalent and valid SQL logical expressions. But the latter is a strange version. 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('transferLog') == None)\
  .show(5)
```

dateTransfer	datetimeTransfer	clientNumber	transferValue	transferCurrency
--------------	------------------	--------------	---------------	------------------

... with 5 more columns: transferID, transferLog, destinationBankNumber  
destinationBankBranch, destinationBankAccount

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

```
transf\
  .where(col('transferLog').isNull())\
  .show(5)
```

dateTransfer	datetimeTransfer	clientNumber	transferValue	transferCurrency
2022-12-31	2022-12-31 14:00:24	5516	7794.31	zing f
2022-12-31	2022-12-31 10:32:07	4965	7919.0	zing f
2022-12-31	2022-12-31 07:37:02	4608	5603.0	dollar \$
2022-12-31	2022-12-31 07:35:05	1121	4365.22	dollar \$
2022-12-31	2022-12-31 02:53:44	1121	4620.0	dollar \$

only showing top 5 rows

... with 5 more columns: transferID, transferLog, destinationBankNumber  
destinationBankBranch, destinationBankAccount

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

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

dateTransfer	datetimeTransfer	clientNumber	transferValue	transferCurrency
2022-12-05	2022-12-05 00:51:00	2197	8240.62	zing f
2022-09-20	2022-09-19 21:59:51	5188	7583.9	dollar \$
2022-09-03	2022-09-03 06:07:59	3795	3654.0	zing f
2022-07-02	2022-07-02 15:29:50	4465	5294.0	dollar \$
2022-06-14	2022-06-14 10:21:55	1121	7302.0	dollar \$

only showing top 5 rows

... with 5 more columns: transferID, transferLog, destinationBankNumber  
destinationBankBranch, destinationBankAccount

## 5.5.6 Filtering dates and datetimes in your DataFrame

Just as a quick side note, when you want to filter rows of your DataFrame that fits into a particular date, you can easily write this particular date as a single string, like in the example below:

```
df_0702 = transf\
  .where(col('dateTransfer') == '2022-07-02')
```

When filtering datetimes you can also write the datetimes as strings, like this:



```

later_0330_pm = transf.where(
    col('datetimeTransfer') > '2022-07-02 03:30:00'
)

```

However, is a better practice to write these particular values using the built-in date and datetime classes of python, like this:

```

from datetime import date, datetime

d = date(2022,2,7)
dt = datetime(2022,2,7,3,30,0)

# Filter all rows where `dateTransfer`
# is equal to "2022-07-02"
df_0702 = transf.where(
    col('dateTransfer') == d
)

# Filter all rows where `datetimeTransfer`
# is greater than "2022-07-02 03:30:00"
later_0330_pm = transf.where(
    col('datetimeTransfer') > dt
)

```

When you translate the above expressions to SQL, you can also write the date and datetime values as strings. However, is also a good idea to use the CAST() SQL function to convert these string values into the correct data type before the actual filter. like this:

```

condition_d = '''
dateTransfer == CAST("2022-07-02" AS DATE)
'''

condition_dt = '''
dateTransfer > CAST("2022-07-02 03:30:00" AS TIMESTAMP)
'''

# Filter all rows where `dateTransfer`
# is equal to "2022-07-02"
df_0702 = transf.where(condition_d)

# Filter all rows where `datetimeTransfer`
# is greater than "2022-07-02 03:30:00"

```

```
later_0330_pm = transf.where(condition_dt)
```

In other words, with the SQL expression `CAST("2022-07-02 03:30:00" AS TIMESTAMP)` we are telling Spark to convert the literal string "2022-07-02 03:30:00" into an actual timestamp value.

### 5.5.7 Searching for a particular pattern in string values

Spark offers different methods to search a particular pattern within a string value. In this section, I want to describe how you can use these different methods to find specific rows in your DataFrame, that fit into the description of these patterns.

#### 5.5.7.1 Starts with, ends with and contains

You can use the column methods `startswith()`, `endswith()` and `contains()`, to search for rows where a input string value starts with, ends with, or, contains a particular pattern, respectively.

These three methods returns a boolean value that indicates if the input string value matched the input pattern that you gave to the method. And you can use these boolean values they return to filter the rows of your DataFrame that fit into the description of these patterns.

Just as an example, in the following code, we are creating a new DataFrame called `persons`, that contains the description of 3 persons (Alice, Bob and Charlie). And I use these three methods to search for different rows in the DataFrame:

```
from pyspark.sql.functions import col

persons = spark.createDataFrame(
    [
        ('Alice', 25),
        ('Bob', 30),
        ('Charlie', 35)
    ],
    ['name', 'age']
)

# Filter the DataFrame to include only rows
# where the "name" column starts with "A"
persons.filter(col('name').startswith('A'))\
    .show()
```

```
+-----+----+
| name|age|
+-----+----+
|Alice| 25|
+-----+----+
```

```
# Filter the DataFrame to include only rows
# where the "name" column ends with "e"
persons.filter(col('name').endsWith('e'))\
.show()
```

```
+-----+----+
|  name|age|
+-----+----+
| Alice| 25|
|Charlie| 35|
+-----+----+
```

```
# Filter the DataFrame to include only rows
# where the "name" column contains "ob"
persons.filter(col('name').contains('ob'))\
.show()
```

```
+-----+----+
|name|age|
+-----+----+
| Bob| 30|
+-----+----+
```

### 5.5.7.2 Using regular expressions or SQL LIKE patterns

In Spark, you can also use a particular “SQL LIKE pattern” or a regular pattern (a.k.a. regex) to filter the rows of a DataFrame, by using the Column methods `like()` and `rlike()`.

In essence, the `like()` method is the pyspark equivalent of the LIKE SQL operator. As a result, this `like()` method expects a SQL pattern as input. This means that you can use the SQL metacharacters `%` (to match any number of characters) and `_` (to match exactly one character) inside this pattern.

```
transf\
.where(col('transferCurrency').like('british%'))\
```

```
.show(5)
```

```
+-----+-----+-----+-----+-----+
|dateTransfer|  datetimeTransfer|clientNumber|transferValue|transferCurrency|
+-----+-----+-----+-----+-----+
|  2022-12-30|2022-12-30 11:30:23|      1455|      5141.0| british pound £|
|  2022-12-30|2022-12-30 02:35:23|      5986|      6076.0| british pound £|
|  2022-12-29|2022-12-29 15:24:04|      4862|      5952.0| british pound £|
|  2022-12-29|2022-12-29 14:16:46|      2197|      8771.0| british pound £|
|  2022-12-29|2022-12-29 06:51:24|      5987|      2345.0| british pound £|
+-----+-----+-----+-----+-----+
```

only showing top 5 rows

... with 5 more columns: transferID, transferLog, destinationBankNumber  
destinationBankBranch, destinationBankAccount

Although the style of pattern matching used by `like()` being very powerful, you might need to use more powerful and flexible patterns. In those cases, you can use the `rlike()` method, which accepts a regular expression as input. In the example below, I am filtering rows where `destinationBankAccount` starts by the characters 54.

```
regex = '^54([0-9]{3})-[0-9]$\ntransf\
.where(col('destinationBankAccount').rlike(regex))\
.show(5)
```

```
+-----+-----+-----+-----+-----+
|dateTransfer|  datetimeTransfer|clientNumber|transferValue|transferCurrency|
+-----+-----+-----+-----+-----+
|  2022-12-29|2022-12-29 02:54:23|      2197|      5752.0| british pound £|
|  2022-12-27|2022-12-27 04:51:45|      4862|     11379.0|      dollar $|
|  2022-12-05|2022-12-05 05:50:27|      4965|      5986.0| british pound £|
|  2022-12-04|2022-12-04 14:31:42|      4965|      8123.0|      dollar $|
|  2022-11-29|2022-11-29 16:23:07|      4862|      8060.0|      zing f|
+-----+-----+-----+-----+-----+
```

only showing top 5 rows

... with 5 more columns: transferID, transferLog, destinationBankNumber  
destinationBankBranch, destinationBankAccount

## 5.6 Selecting a subset of rows from your DataFrame

At some point, you might need to use just a small piece of your DataFrame over the next steps of your pipeline, and not the entire thing. For example, you may want to select just the first (or last) 5 rows of this DataFrame, or, maybe, you need to take a random sample of rows from it.

In this section I will discuss the main methods offered by Spark to deal with these scenarios. Each method returns a subset of rows from the original DataFrame as a result. But each method works differently from the other, and uses a different strategy to retrieve this subset.

### 5.6.1 Limiting the number of rows in your DataFrame

The `limit()` method is very similar to the `LIMIT` SQL keyword. It limits the number of rows present in your DataFrame to a specific amount. So, if I run `transf.limit(1)` I get a new DataFrame as a result, which have only a single row from the `transf` DataFrame. As you can see below:

```
single_transfer = transf.limit(1)
single_transfer.show()
```

```
+-----+-----+-----+-----+-----+
|dateTransfer|  datetimeTransfer|clientNumber|transferValue|transferCurrency|
+-----+-----+-----+-----+-----+
|  2022-12-31|2022-12-31 14:00:24|      5516|    7794.31|          zing f|
+-----+-----+-----+-----+-----+
... with 5 more columns: transferID, transferLog, destinationBankNumber
                        destinationBankBranch, destinationBankAccount
```

Is worth mentioning that the `limit()` method will always try to limit your original DataFrame, to the first  $n$  rows. This means that the command `df.limit(430)` tries to limit the `df` DataFrame to its first 430 rows.

This also means that 430 is the maximum number of rows that will be taken from the `df` DataFrame. So, if `df` DataFrame has less than 430 lines, like 14 rows, than, nothing will happen, i.e. the result of `df.limit(430)` will be equivalent to the `df` DataFrame itself.

### 5.6.2 Getting the first/last $n$ rows of your DataFrame

The methods `head()` and `tail()` allows you to collect the first/last  $n$  rows of your DataFrame, respectively. One key aspect from these methods, is that they return a list of Row values, instead of a new DataFrame (such as the `limit()` method). You can compare these methods to the `take()` and

`collect()` methods that we introduced at Section 5.2, because they both produce a list of Row values as well.

Now, the `head()` method produce the same output as the `take()` method. However, these two methods work very differently under the hoods, and, are recommended to be used in different scenarios.

More specifically, if you have a big DataFrame (i.e. a DataFrame with many rows) is recommended to use `take()` (instead of `head()`) to collect the first  $n$  rows from it. Because the `head()` method makes Spark to load the entire DataFrame into the driver's memory, and this can easily cause an “out of memory” situation for big DataFrames. So, use the `head()` method only for small DataFrames.

In the example below, we are using these methods to get the first and last 2 rows of the `transf` DataFrame:

```
# First 2 rows of `transf` DataFrame:
first2 = transf.head(2)
# Last 2 rows of `transf` DataFrame:
last2 = transf.tail(2)

print(last2)
```

```
[Row(dateTransfer=datetime.date(2022, 1, 1),
datetimeTransfer=datetime.datetime(2022, 1, 1,
4, 7, 44),
clientNumber=5987, transferValue=8640.06, transferCurrency='dollar $',
transferID=20221144, transferLog=None, destinationBankNumber=666,
destinationBankBranch=6552, destinationBankAccount='70021-4'), Row(dateTransfer=
datetime.date(2022,
1, 1), datetimeTransfer=datetime.datetime(2022,
1, 1, 3,
56, 58), clientNumber=6032,
transferValue=5076.61, transferCurrency='dollar $', transferID=20221143,
transferLog=None, destinationBankNumber=33, destinationBankBranch=8800,
destinationBankAccount='41326-5')]
```

### 5.6.3 Taking a random sample of your DataFrame

With the `sample()` you can take a random sample of rows from your DataFrame. In other words, this method returns a new DataFrame with a random subset of rows from the original DataFrame.

This method have three main arguments, which are:

- `withReplacement`: a boolean value indicating if the samples are with replacement or not. Defaults to `False`;

- `fraction`: the fraction of rows you want to sample from the DataFrame. Have to be a positive float value, from 0 to 1;
- `seed`: an integer representing the seed for the sampling process. This is an optional argument;

In the example below, we are trying to get a sample that represents 15% of the original `transf` DataFrame, and using the integer 24 as our sampling seed:

```
transf_sample = transf.sample(fraction = 0.15, seed = 24)
transf_sample.show(5)
```

```
+-----+-----+-----+-----+-----+
|dateTransfer|  datetimeTransfer|clientNumber|transferValue|transferCurrency|
+-----+-----+-----+-----+-----+
|  2022-12-31|2022-12-31 01:02:06|      4862|      6714.0|      dollar $|
|  2022-12-30|2022-12-30 00:18:25|      5832|      6333.0|      dollar $|
|  2022-12-29|2022-12-29 06:51:24|      5987|      2345.0| british pound £|
|  2022-12-27|2022-12-27 14:08:01|      3294|      6617.17|      dollar $|
|  2022-12-26|2022-12-26 11:25:09|      5832|      8178.0|      euro €|
+-----+-----+-----+-----+-----+
```

only showing top 5 rows

... with 5 more columns: `transferID`, `transferLog`, `destinationBankNumber`, `destinationBankBranch`, `destinationBankAccount`

In other words, the `fraction` argument represents a fraction of the total number of rows in the original DataFrame. Since the `transf` DataFrame have 2421 rows in total, by setting the `fraction` argument to 0.15, we are asking Spark to collect a sample from `transf` that have approximately  $0.15 \times 2421 \approx 363$  rows.

If we calculate the number of rows in `transf_sample` DataFrame, we can see that this DataFrame have a number of rows close to 363:

```
transf_sample.count()
```

355

Furthermore, the sampling seed is just a way to ask Spark to produce the same random sample of the original DataFrame. That is, the sampling seed makes the result sample fixed. You always get the same random sample when you run the `sample()` method.

On the other hand, when you do not set the `seed` argument, then, Spark will likely produce a different random sample of the original DataFrame every time you run the `sample()` method.

## 5.7 Managing the 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', 'datetimeTransfer',  
    'clientNumber', 'transferValue'  
  )\  
  .show(5)
```

```
+-----+-----+-----+-----+  
|transferID|  datetimeTransfer|clientNumber|transferValue|  
+-----+-----+-----+-----+  
|  20223563|2022-12-31 14:00:24|      5516|      7794.31|  
|  20223562|2022-12-31 10:32:07|      4965|       7919.0|  
|  20223561|2022-12-31 07:37:02|      4608|       5603.0|  
|  20223560|2022-12-31 07:35:05|      1121|       4365.22|  
|  20223559|2022-12-31 02:53:44|      1121|       4620.0|  
+-----+-----+-----+-----+
```

only showing top 5 rows

### 5.7.1 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.

For example, you might 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, you need to use it after a `col()` or `column()` function, or, after a column name using the dot operator.

```
transf\  
  .select(  
    'datetimeTransfer',  
    col('transferID').alias('ID_of_transfer'),  
    transf.clientNumber.alias('clientID')  
  )\  
  .show(5)
```



```
.show(5)
```

```
+-----+-----+-----+
| datetimeTransfer|ID_of_transfer|clientID|
+-----+-----+-----+
|2022-12-31 14:00:24|      20223563|    5516|
|2022-12-31 10:32:07|      20223562|    4965|
|2022-12-31 07:37:02|      20223561|    4608|
|2022-12-31 07:35:05|      20223560|    1121|
|2022-12-31 02:53:44|      20223559|    1121|
+-----+-----+-----+
```

only showing top 5 rows

By using this `alias()` method, you can rename multiple columns within a single `select()` call. However, you can use the `withColumnRenamed()` method to rename just a single column of your `DataFrame`. The first argument of this method, is the current name of this column, and, the second argument, is the new name of this column.

```
transf\
  .withColumnRenamed('clientNumber', 'clientID')\
  .show(5)
```

```
+-----+-----+-----+-----+-----+
|dateTransfer| datetimeTransfer|clientID|transferValue|transferCurrency|
+-----+-----+-----+-----+-----+
| 2022-12-31|2022-12-31 14:00:24|    5516|      7794.31|          zing f|
| 2022-12-31|2022-12-31 10:32:07|    4965|      7919.0|          zing f|
| 2022-12-31|2022-12-31 07:37:02|    4608|      5603.0|        dollar $|
| 2022-12-31|2022-12-31 07:35:05|    1121|      4365.22|        dollar $|
| 2022-12-31|2022-12-31 02:53:44|    1121|      4620.0|        dollar $|
+-----+-----+-----+-----+-----+
```

only showing top 5 rows

... with 5 more columns: transferID, transferLog, destinationBankNumber  
destinationBankBranch, destinationBankAccount

## 5.7.2 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('dateTransfer', 'clientNumber')\
  .show(5)
```

```
+-----+-----+-----+-----+-----+
|  datetimeTransfer|transferValue|transferCurrency|transferID|transferLog|
+-----+-----+-----+-----+-----+
|2022-12-31 14:00:24|      7794.31|           zing f| 20223563|      NULL|
|2022-12-31 10:32:07|      7919.0|           zing f| 20223562|      NULL|
|2022-12-31 07:37:02|      5603.0|        dollar $| 20223561|      NULL|
|2022-12-31 07:35:05|      4365.22|        dollar $| 20223560|      NULL|
|2022-12-31 02:53:44|      4620.0|        dollar $| 20223559|      NULL|
+-----+-----+-----+-----+-----+
```

only showing top 5 rows

... with 3 more columns: destinationBankNumber, destinationBankBranch  
destinationBankAccount

### 5.7.3 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 a column to a specific data type, you can use `cast()` or `astype()` methods inside `select()`. The `astype()` method is just an alias to `cast()`. This `cast()` method is very similar to the `CAST()` function in SQL, and belongs to the `Column` class, so, you should always use it after a column name with the dot operator, or, a `col()/column()` function:

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

```
+-----+-----+-----+
|transferValue|value_as_integer|value_as_boolean|
+-----+-----+-----+
|      7794.31|           7794|           true|
|      7919.0|           7919|           true|
+-----+-----+-----+
```

	5603.0	5603	true
	4365.22	4365	true
	4620.0	4620	true

+-----+-----+-----+

only showing top 5 rows

To use `cast()` or `astype()` methods, 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()` or `astype()` 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()`;

#### 5.7.4 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(
    'transferValue',
    (col('transferValue') / 1000).alias('by_1000')
  )\
  .show(5)
```

+-----+-----+
transferValue by_1000
+-----+-----+
7794.31 7.79431
7919.0 7.919
5603.0 5.603
4365.22 4.36522
4620.0 4.62

```
+-----+-----+
only showing top 5 rows
```

This `by_1000` column do not exist in `transf` DataFrame. It was calculated and added to the final result by `select()`. The formula `col('transferValue') / 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('transferValue') / 1000).alias('by_1000')
)\
.show(5)
```

```
+-----+-----+-----+-----+-----+
|dateTransfer|  datetimeTransfer|clientNumber|transferValue|transferCurrency|
+-----+-----+-----+-----+-----+
|  2022-12-31|2022-12-31 14:00:24|      5516|      7794.31|          zing f|
|  2022-12-31|2022-12-31 10:32:07|      4965|       7919.0|          zing f|
|  2022-12-31|2022-12-31 07:37:02|      4608|       5603.0|        dollar $|
|  2022-12-31|2022-12-31 07:35:05|      1121|       4365.22|        dollar $|
|  2022-12-31|2022-12-31 02:53:44|      1121|       4620.0|        dollar $|
+-----+-----+-----+-----+-----+
only showing top 5 rows
... with 6 more columns: transferID, transferLog, destinationBankNumber
                        destinationBankBranch, destinationBankAccount, by_1000
```

## 5.8 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 the equation) that represents this new column. As an example, I could reproduce the same `by_1000` column like this:

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

```
+-----+-----+-----+-----+-----+  
|dateTransfer|  datetimeTransfer|clientNumber|transferValue|transferCurrency|  
+-----+-----+-----+-----+-----+  
|  2022-12-31|2022-12-31 14:00:24|      5516|      7794.31|      zing f|  
|  2022-12-31|2022-12-31 10:32:07|      4965|      7919.0|      zing f|  
|  2022-12-31|2022-12-31 07:37:02|      4608|      5603.0|    dollar $|  
|  2022-12-31|2022-12-31 07:35:05|      1121|      4365.22|    dollar $|  
|  2022-12-31|2022-12-31 02:53:44|      1121|      4620.0|    dollar $|  
+-----+-----+-----+-----+-----+
```

only showing top 5 rows

... with 6 more columns: transferID, transferLog, destinationBankNumber  
destinationBankBranch, destinationBankAccount, by\_1000

A lot of the times we use the functions from `pyspark.sql.functions` module to produce such formulas used by `withColumn()`. You can checkout the complete list of functions present in this module, by visiting the official documentation of pyspark<sup>2</sup>.

You will see a lot more examples of formulas and uses of `withColumn()` throughout this book. For now, 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.

## 5.9 Sorting rows of your DataFrame

Spark, or, pyspark, provides the `orderBy()` and `sort()` `DataFrame` method to sort rows. They both work the same way: you just 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 `transferValue` column. By default, Spark uses an ascending order while sorting your rows.

---

<sup>2</sup><https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql.html#functions>

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

dateTransfer	datetimeTransfer	clientNumber	transferValue	transferCurrency
2022-07-22	2022-07-22 16:06:25	3795	60.0	dollar \$
2022-05-09	2022-05-09 14:02:15	3284	104.0	dollar \$
2022-09-16	2022-09-16 20:35:40	3294	129.09	zing f
2022-12-18	2022-12-18 08:45:30	1297	142.66	dollar \$
2022-08-20	2022-08-20 09:27:55	2727	160.0	dollar \$

only showing top 5 rows

... with 5 more columns: transferID, transferLog, destinationBankNumber  
destinationBankBranch, destinationBankAccount

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 clientNumber column first, then, is going to sort the rows of each clientNumber according to transferValue column.

```
transf\
  .orderBy('clientNumber', 'transferValue')\
  .show(5)
```

dateTransfer	datetimeTransfer	clientNumber	transferValue	transferCurrency
2022-03-30	2022-03-30 11:57:22	1121	461.0	euro €
2022-05-23	2022-05-23 11:51:02	1121	844.66	british pound £
2022-08-24	2022-08-24 13:51:30	1121	1046.93	euro €
2022-09-23	2022-09-23 19:49:19	1121	1327.0	british pound £
2022-06-25	2022-06-25 17:07:08	1121	1421.0	dollar \$

only showing top 5 rows

... with 5 more columns: transferID, transferLog, destinationBankNumber  
destinationBankBranch, destinationBankAccount

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 clientNumber column using

an ascending order. However, it will use the values from transferValue column in a descending order to sort the rows in each clientNumber.

```
transf\  
  .orderBy('clientNumber', col('transferValue').desc())\  
  .show(5)
```

```
+-----+-----+-----+-----+-----+  
|dateTransfer|  datetimeTransfer|clientNumber|transferValue|transferCurrency|  
+-----+-----+-----+-----+-----+  
|  2022-08-18|2022-08-18 13:57:12|      1121|    11490.37|          zing f|  
|  2022-11-05|2022-11-05 08:00:37|      1121|    10649.59|          dollar $|  
|  2022-05-17|2022-05-17 10:27:05|      1121|    10471.23| british pound £|  
|  2022-05-15|2022-05-15 00:25:49|      1121|     10356.0|          dollar $|  
|  2022-06-10|2022-06-09 23:51:39|      1121|     10142.0|          dollar $|  
+-----+-----+-----+-----+-----+
```

only showing top 5 rows

... with 5 more columns: transferID, transferLog, destinationBankNumber  
destinationBankBranch, destinationBankAccount

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.

## 5.10 Calculating aggregates

To calculate aggregates of a Spark DataFrame we have two main paths: 1) we can use some standard DataFrame methods, like count() or sum(), to calculate a single aggregate; 2) or, we can use the agg() method to calculate multiple aggregates at the same time.

### 5.10.1 Using standard DataFrame methods

The Spark DataFrame class by itself provides a single aggregate method, which is count(). With this method, you can find out how many rows your DataFrame have. In the example below, we can see that transf have 2421 rows.

```
transf.count()
```

2421

However, if you have a **grouped** DataFrame (we will learn more about these objects very soon), pyspark provides some more aggregate methods, which are listed below:

- `mean()`: calculate the average value of each numeric column;
- `sum()`: return the total sum of a column;
- `count()`: count the number of rows;
- `max()`: compute the maximum value of a column;
- `min()`: compute the minimum value of a column;

This means that you can use any of the above methods after a `groupby()` call, to calculate aggregates *per group* in your Spark DataFrame. For now, let's forget about this “groupby” detail, and learn how to calculate different aggregates by using the `agg()` method.

### 5.10.2 Using the `agg()` method

With the `agg()` method, we can calculate many different aggregates at the same time. In this method, you should provide an expression that describes what aggregate measure you want to calculate.

In most cases, this “aggregate expression” will be composed of functions from the `pyspark.sql.functions` module. So, having familiarity with the functions present in this module will help you to compose the formulas of your aggregations in `agg()`.

In the example below, I am using the `sum()` and `mean()` functions from `pyspark.sql.functions` to calculate the total sum and the total mean of the `transferValue` column in the `transf` DataFrame. I am also using the `countDistinct()` function to calculate the number of distinct values in the `clientNumber` column.

```
import pyspark.sql.functions as F

transf.agg(
    F.sum('transferValue').alias('total_value'),
    F.mean('transferValue').alias('mean_value'),
    F.countDistinct('clientNumber').alias('number_of_clients')
)\
.show()
```

```
+-----+-----+-----+
|      total_value|      mean_value|number_of_clients|
+-----+-----+-----+
|1.5217690679999998E7|6285.704535315985|                26|
+-----+-----+-----+
```



### 5.10.3 Without groups, we calculate a aggregate of the entire DataFrame

When we do not define any group for the input DataFrame, `agg()` always produce a new DataFrame with one single row (like in the above example). This happens because we are calculating aggregates of the entire DataFrame, that is, a set of single values (or single measures) that summarizes (in some way) the entire DataFrame.

In the other hand, when we define groups in a DataFrame (by using the `groupby()` method), the calculations performed by `agg()` are made inside each group in the DataFrame. In other words, instead of summarizing the entire DataFrame, `agg()` will produce a set of single values that describes (or summarizes) each group in the DataFrame.

This means that each row in the resulting DataFrame describes a specific group in the original DataFrame, and, `agg()` usually produces a DataFrame with more than one single row when its calculations are performed by group. Because our DataFrames usually have more than one single group.

### 5.10.4 Calculating aggregates per group in your DataFrame

But how you define the groups inside your DataFrame? To do this, we use the `groupby()` and `groupBy()` methods. These methods are both synonymous (they do the same thing).

These methods, produce a **grouped** DataFrame as a result, or, in more technical words, a object of class `pyspark.sql.group.GroupedData`. You just need to provide, inside this `groupby()` method, the name of the columns that define (or “mark”) your groups.

In the example below, I am creating a grouped DataFrame per client defined in the `clientNumber` column. This means that each distinct value in the `clientNumber` column defines a different group in the DataFrame.

```
transf_per_client = transf.groupBy('clientNumber')
transf_per_client
```

```
GroupedData[grouping expressions: [clientNumber], value: [dateTransfer: date, da
tetimeTransfer: timestamp ... 8 more fields], type: GroupBy]
```

At first, it appears that nothing has changed. But the `groupBy()` method always returns a new object of class `pyspark.sql.group.GroupedData`. As a consequence, we can no longer use some of the DataFrame methods that we used before, like the `show()` method to see the DataFrame.

That's ok, because we usually do not want to keep this grouped DataFrame for much time. This grouped DataFrame is just a passage (or a bridge) to the result we want, which is, to calculate aggregates **per group** of the DataFrame.

As an example, I can use the `max()` method, to find out which is the highest value that each user have tranfered, like this:

```
transf_per_client\
  .max('transferValue')\
  .show(5)
```

```
+-----+-----+
|clientNumber|max(transferValue)|
+-----+-----+
|      1217|      12601.0|
|      2489|      12644.56|
|      3284|      12531.84|
|      4608|      10968.31|
|      1297|      11761.0|
+-----+-----+
```

only showing top 5 rows

Remember that, by using standard DataFrame methods (like `max()` in the example above) we can calculate only a single aggregate value. But, with `agg()` we can calculate more than one aggregate value at the same time. Since our `transf_per_client` object is a **grouped** DataFrame, `agg()` will calculate these aggregates per group.

As an example, if I apply `agg()` with the exact same expressions exposed on Section 5.10.2 with the `transf_per_client` DataFrame, instead of a DataFrame with one single row, I get a new DataFrame with nine rows. In each row, I have the total and mean values for a specific user in the input DataFrame.

```
transf_per_client\
  .agg(
    F.sum('transferValue').alias('total_value'),
    F.mean('transferValue').alias('mean_value')
  )\
  .show(5)
```

```
+-----+-----+-----+
|clientNumber|total_value|mean_value|
+-----+-----+-----+
|      1217|575218.20999999998| 6185.142043010751|
|      2489|546543.09000000001| 6355.152209302327|
|      3284|581192.57000000001| 6054.089270833334|
|      4608|      448784.44| 6233.117222222222|
```

```
|          1297|594869.6699999999|6196.5590624999995|
+-----+-----+-----+
only showing top 5 rows
```

## 6 Importing data to Spark

Another way of creating Spark DataFrames, is to read (or import) data from somewhere and convert it to a Spark 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 ODBC/JDBC connections.

To read (or import) any data to Spark, we use a “read engine”, and there are many different read engines available in Spark. Each engine is used to read a specific file format, or to import data from a specific type of data source, and we access these engines by using the read module from your Spark Session object.

### 6.1 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 the other machine, or download it from the internet.

But in order to Spark read any type of static file stored inside your computer, **it always need to know the path to this file**. Every OS have its own file system, and every file in your computer is stored in a specific address of this file system. The “path” to this file is the path (or “steps”) that your computer needs to follow to reach this specific address, where the file is stored.

As we pointed out earlier, to read any static file in Spark, you use one of the available “read engines”, which are in the `spark.read` module of your Spark Session. This means that, each read engine in this module is responsible for reading a specific file format.

If you want to read a CSV file for example, you use the `spark.read.csv()` engine. In contrast, if you want to read a JSON file, you use the `spark.read.json()` engine instead. But no matter what read engine you use, you always give the path to your file to any of these “read engines”.

The main read engines available in Spark are:

- `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.
- `spark.read.text()`: to read text files.
- `spark.read.jdbc()`: to read data from databases using the JDBC API.

For example, to read a JSON file called `sales.json` that is stored in my Data folder, I can do this:

```
json_data = spark.read.json("../Data/sales.json")
json_data.show()
```

```
+-----+-----+-----+-----+-----+-----+
|price|product_id|product_name|sale_id|          timestamp|units|
+-----+-----+-----+-----+-----+-----+
| 3.12|      134| Milk 1L Mua| 328711|2022-02-01T22:10:02|    1|
| 1.22|      110|   Coke 350ml| 328712|2022-02-03T11:42:09|    3|
| 4.65|      117|    Pepsi 2L| 328713|2022-02-03T14:22:15|    1|
| 1.22|      110|   Coke 350ml| 328714|2022-02-03T18:33:08|    1|
| 0.85|      341|Trident Mint| 328715|2022-02-04T15:41:36|    1|
+-----+-----+-----+-----+-----+-----+
```

## 6.2 An example with a CSV 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 static file, Spark needs to know the path to this `people.csv` file. In other words, Spark needs to know where this file is stored in my computer, to be able to read it.

In my specific case, considering where this `Data` folder is in my computer, a relative path to it would be `"../Data/"`. Having the path to the folder where `people.csv` is stored, I just need to add this file to the path, resulting in `"../Data/people.csv"`. See in the example below, that I gave this path to the `read.csv()` method of my Spark Session. As a result, Spark will visit this address, and, read the file that is stored there:

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

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

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

```
+-----+-----+
| _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<sup>1</sup> 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\..."`.

```
# The absolute path to `people.csv`:
path = "/home/pedro/Documents/Projects/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|
+-----+-----+
```

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/Projects/Books/Introd-pyspark/weird-file.csv
```

---

<sup>1</sup>That is, the complete path to the file, or, in other words, a path that starts in the root folder of your hard drive.

Every time you face this “Path does not exist” error, it means that Spark did not find the file that you described in the path you gave to `spark.read`. In this case, is very likely that you have a typo or a mistake in your path. Maybe you forgot to add the `.csv` extension to the name of your file. Or maybe you forgot to use the right angled slash (/) instead of the left angled slash (\). Or maybe, you gave the path to folder *x*, when in fact, you wanted to reach the folder *y*.

Sometimes, is useful to list all the files that are stored inside the folder you are trying to access. This way, you can make sure that you are looking at the right folder of your file system. To do that, you can use the `listdir()` function from `os` module of python. As an example, I can list all the files that are stored inside of the `Data` folder in this way:

```
from os import listdir
listdir("../Data/")
```

```
['accounts.csv',
 'books.txt',
 'livros.txt',
 'logs.json',
 'penguins.csv',
 'people.csv',
 'sales.json',
 'transf.csv',
 'transf_reform.csv',
 'user-events.json']
```

## 6.3 Import options

While reading and importing data from any type of data source, Spark will always use the default values for each import option defined by the read engine you are using, unless you explicit ask it to use a different value. 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 (like 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 correctly 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<sup>2</sup>.

To define, or, set a specific import option, you use the `option()` method from a `DataFrameReader` object. To produce this kind of object, you use the `spark.read` module, like in the example below. Each call to the `option()` method is used to set a single import option.

Notice that the “read engine” of Spark (i.e. `csv()`) is the last method called at this chain (or sequence) of steps. In other words, you start by creating a `DataFrameReader` object, then, set the import options, and lastly, you define which “read engine” you want to use.

```
# Creating a `DataFrameReader` object:
df_reader = spark.read
# Setting the import options:
df_reader = df_reader\
    .option("sep", "$")\
    .option("locale", "pt-BR")

# Setting the "read engine" to be used with `.csv()`:
my_data = df_reader\
    .csv("../Data/a-csv-file.csv")
```

If you prefer, you can also merge all these calls together like this:

```
spark.read\ # a `DataFrameReader` object
    .option("sep", "$")\ # Setting the `sep` option
    .option("locale", "pt-BR")\ # Setting the `locale` option
    .csv("../Data/a-csv-file.csv") # The "read engine" to be used
```

There are many different import options for each read engine, and you can see the full list in the official documentation for Spark<sup>3</sup>. But let's just give you a brief overview of the probably most popular import options:

- `sep`: sets the separator character for each field and value in the CSV file (defaults to `" , "`);
- `encoding`: sets the character encoding of the file to be read (defaults to `"UTF-8"`);

---

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

<sup>3</sup><https://spark.apache.org/docs/latest/sql-data-sources-csv.html>



- `header`: boolean (defaults to `False`), should Spark consider the first line of the file as the header of the `DataFrame` (i.e. the name of the columns) ?
- `dateFormat` and `timestampFormat`: sets the format for dates and timestamps in the file (defaults to `"yyyy-MM-dd"` and `"yyyy-MM-dd'T'HH:mm:ss[.SSS][XXX]"` respectively);

## 6.4 Setting the separator character for CSV files

In this section, we will use the `transf_reform.csv` file to demonstrate how to set the separator character of a CSV file. This file, contains some data of transfers made in a fictitious bank. Is worth mentioning that this `sep` import option is only available for CSV files.

Lets use the `peek_file()` function defined below to get a quick peek at the first 5 lines of this file. If you look closely to these lines, you can identify that this CSV files uses the `";"` character to separate fields and values, and not the american standard `" , "` character.

```
def peek_file(path, n_lines = 5):
    with open(path) as file:
        lines = [next(file) for i in range(n_lines)]
        text = ''.join(lines)
        print(text)

peek_file("../Data/transf_reform.csv")
```

```
datetime;user;value;transferid;country;description
2018-12-06T22:19:19Z;Eduardo;598.5984;116241629;Germany;
2018-12-06T22:10:34Z;Júlio;4610.955;115586504;Germany;
2018-12-06T21:59:50Z;Nathália;4417.866;115079280;Germany;
2018-12-06T21:54:13Z;Júlio;2739.618;114972398;Germany;
```

This is usually the standard adopted by countries that uses a comma to define decimal places in real numbers. In other words, in some countries, the number `3.45` is usually written as `3,45`.

Anyway, we know now that the `transf_reform.csv` file uses a different separator character, so, to correctly read this CSV file into Spark, we need to set the `sep` import option. Since this file comes with the column names in the first line, I also set the `header` import option to read this first line as the column names as well.

```
transf = spark.read\
    .option("sep", ";")\
    .option("header", True)\
    .csv("../Data/transf_reform.csv")
```

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

```
+-----+-----+-----+-----+-----+-----+
|      datetime|   user|   value|transferid|country|description|
+-----+-----+-----+-----+-----+-----+
|2018-12-06T22:19:19Z| Eduardo|598.5984| 116241629|Germany|      NULL|
|2018-12-06T22:10:34Z|   Júlio|4610.955| 115586504|Germany|      NULL|
|2018-12-06T21:59:50Z|Nathália|4417.866| 115079280|Germany|      NULL|
|2018-12-06T21:54:13Z|   Júlio|2739.618| 114972398|Germany|      NULL|
|2018-12-06T21:41:27Z|   Ana|1408.261| 116262934|Germany|      NULL|
+-----+-----+-----+-----+-----+-----+
```

only showing top 5 rows

## 6.5 Setting the encoding of the file

Spark will always assume that your static files use the UTF-8 encoding system. But, that might not be the case for your specific file. In this situation, you have to tell Spark which is the appropriate encoding system to be used while reading the file. This encoding import option is available both for CSV and JSON files.

To do this, you can set the encoding import option, with the name of the encoding system to be used. As an example, let's look at the file `books.txt`, which is a CSV file encoded with the ISO-8859-1 system (i.e. the Latin 1 system).

If we use the defaults in Spark, you can see in the result below that some characters in the Title column are not correctly interpreted. Remember, this problem occurs because of a mismatch in encoding systems. Spark thinks `books.txt` is using the UTF-8 system, but, in reality, it uses the ISO-8859-1 system:

```
books = spark.read\
    .option("header", True)\
    .csv("../Data/books.txt")

books.show()
```

```
+-----+-----+-----+
|      Title|      Author| Price|
+-----+-----+-----+
|      O Hobbit| J. R. R. Tolkien| 40.72|
|Matemática para E...|Carl P. Simon and...|139.74|
```

```
|Microeconomia: um...|      Hal R. Varian| 141.2|
|      A Luneta mbar|      Philip Pullman| 42.89|
+-----+-----+-----+
```

But if we tell Spark to use the ISO-8859-1 system while reading the file, then, all problems are solved, and all characters in the file are correctly interpreted, as you see in the result below:

```
books = spark.read\
    .option("header", True)\
    .option("encoding", "ISO-8859-1")\
    .csv("../Data/books.txt")

books.show()
```

```
+-----+-----+-----+
|          Title|          Author| Price|
+-----+-----+-----+
|      O Hobbit|      J. R. R. Tolkien| 40.72|
|Matemática para E...|Carl P. Simon and...|139.74|
|Microeconomia: um...|      Hal R. Varian| 141.2|
|      A Luneta Âmbar|      Philip Pullman| 42.89|
+-----+-----+-----+
```

## 6.6 Setting the format of dates and timestamps

The format that humans write dates and timestamps vary drastically over the world. By default, Spark will assume that the dates and timestamps stored in your file are in the format described by the ISO-8601 standard. That is, the “YYYY-mm-dd”, or, “year-month-day” format.

But this standard might not be the case for your file. For example: the brazilian people usually write dates in the format “dd/mm/YYYY”, or, “day/month/year”; some parts of Spain write dates in the format “YYYY/dd/mm”, or, “year/day/month”; on Nordic countries (i.e. Sweden, Finland) dates are written in “YYYY.mm.dd” format.

Every format of a date or timestamp is defined by using a string with the codes of each part of the date/timestamp, like the letter ‘Y’ which represents a 4-digit year, or the letter ‘d’ which represents a 2-digit day. You can see the complete list of codes and their description in the official documentation of Spark<sup>4</sup>.

<sup>4</sup><https://spark.apache.org/docs/latest/sql-ref-datetime-pattern.html>

As an example, let's look into the `user-events.json` file. We can see that the dates and timestamps in this file are using the “dd/mm/YYYY” and “dd/mm/YYYY HH:mm:ss” formats respectively.

```
peek_file("../Data/user-events.json", n_lines=3)
```

```
{"dateOfEvent": "15/06/2022", "timeOfEvent": "15/06/2022 14:33:10", "userId": "b902e51e-d043-4a66-afc4-a820173e1bb4", "nameOfEvent": "entry"}
{"dateOfEvent": "15/06/2022", "timeOfEvent": "15/06/2022 14:40:08", "userId": "b902e51e-d043-4a66-afc4-a820173e1bb4", "nameOfEvent": "click: shop"}
{"dateOfEvent": "15/06/2022", "timeOfEvent": "15/06/2022 15:48:41", "userId": "b902e51e-d043-4a66-afc4-a820173e1bb4", "nameOfEvent": "select: payment-method"}
```

Date variables are usually interpreted by Spark as string variables. In other words, Spark usually does not convert data that contains dates to the date type of Spark. In order to Spark

```
from pyspark.sql.types import StructType, StructField
from pyspark.sql.types import DateType, StringType, TimestampType

schema = StructType([
    StructField('dateOfEvent', DateType(), True),
    StructField('timeOfEvent', TimestampType(), True),
    StructField('userId', StringType(), True),
    StructField('nameOfEvent', StringType(), True)
])

user_events = spark.read\
    .option("dateFormat", "d/M/y")\
    .option("timestampFormat", "d/M/y k:m:s")\
    .json("../Data/user-events.json", schema = schema)

user_events.show()
```

```
+-----+-----+-----+-----+
|dateOfEvent|timeOfEvent|userId|nameOfEvent|
+-----+-----+-----+-----+
| 2022-06-15|2022-06-15 14:33:10|b902e51e-d043-4a6...|entry|
| 2022-06-15|2022-06-15 14:40:08|b902e51e-d043-4a6...|click: shop|
| 2022-06-15|2022-06-15 15:48:41|b902e51e-d043-4a6...|select: payment-m...|
+-----+-----+-----+-----+
```

## 7 Working with SQL in pyspark

As we discussed in Chapter 2, Spark is a **multi-language** engine for large-scale data processing. This means that we can build our Spark application using many different languages (like Java, Scala, Python and R). Furthermore, you can also use the Spark SQL module of Spark to translate all of your transformations into pure SQL queries.

In more details, Spark SQL is a Spark module for structured data processing (*Apache Spark Official Documentation* 2022). Because this module works with Spark DataFrames, using SQL, you can translate all transformations that you build with the DataFrame API into a SQL query.

Therefore, you can mix python code with SQL queries very easily in Spark. Virtually all transformations exposed in python throughout this book, can be translated into a SQL query using this module of Spark. We will focus a lot on this exchange between Python and SQL in this chapter.

However, this also means that the Spark SQL module does not handle the transformations produced by the unstructured APIs of Spark, i.e. the Dataset API. Since the Dataset API is not available in pyspark, it is not covered in this book.

### 7.1 The sql() method as the main entrypoint

The main entrypoint, that is, the main bridge that connects Spark SQL to Python is the sql() method of your Spark Session. This method accepts a SQL query inside a string as input, and will always output a new Spark DataFrame as result. That is why I used the show() method right after sql(), in the example below, to see what this new Spark DataFrame looked like.

As a first example, lets look at a very basic SQL query, that just select a list of code values:

```
SELECT *  
FROM (  
  VALUES (11), (31), (24), (35)  
) AS List(Codes)
```

To run the above SQL query, and see its results, I must write this query into a string, and give this string to the sql() method of my Spark Session. Then, I use the show() action to see the actual result rows of data generated by this query:

```

sql_query = '''
SELECT *
FROM (
    VALUES (11), (31), (24), (35)
) AS List(Codes)
'''

spark.sql(sql_query).show()

```

```

+-----+
|Codes|
+-----+
|   11|
|   31|
|   24|
|   35|
+-----+

```

If you want to execute a very short SQL query, is fine to write it inside a single pair of quotation marks (for example "SELECT \* FROM sales.per\_day"). However, since SQL queries usually take multiple lines, you can write your SQL query inside a python docstring (created by a pair of three quotation marks), like in the example above.

Having this in mind, every time you want to execute a SQL query, you can use this `sql()` method from the object that holds your Spark Session. So the `sql()` method is the bridge between pyspark and SQL. You give it a pure SQL query inside a string, and, Spark will execute it, considering your Spark SQL context.

### 7.1.1 A single SQL statement per run

Is worth pointing out that, although being the main bridge between Python and SQL, the Spark Session `sql()` method can execute only a single SQL statement per run. This means that if you try to execute two sequential SQL statements at the same time with `sql()`, then, Spark SQL will automatically raise a `ParseException` error, which usually complains about an “extra input”.

In the example below, we are doing two very basic steps to SQL. We first create a dummy database with a `CREATE DATABASE` statement, then, we ask SQL to use this new database that we created as the default database of the current session, with a `USE` statement.

```

CREATE DATABASE `dummy`;
USE `dummy`;

```

If we try to execute these two steps at once, by using the `sql()` method, Spark complains with a `ParseException`, indicating that we have a syntax error in our query, like in the example below:

```
query = '''
CREATE DATABASE `dummy`;
USE `dummy`;
'''

spark.sql(query).show()
```

Traceback (most recent call last):

```
File "<stdin>", line 1, in <module>
File "/opt/spark/python/pyspark/sql/session.py", line 1034, in sql
    return DataFrame(self._jsparkSession.sql(sqlQuery), self)
File "/opt/spark/python/lib/py4j-0.10.9.5-src.zip/py4j/java_gateway.py", line
1321, in __call__
File "/opt/spark/python/pyspark/sql/utils.py", line 196, in deco
    raise converted from None
pyspark.sql.utils.ParseException:
Syntax error at or near 'USE': extra input 'USE'(line 3, pos 0)
```

== SQL ==

```
CREATE DATABASE `dummy`;
USE `dummy`;
^^^
```

However, there is nothing wrong about the above SQL statements. They are both correct and valid SQL statements, both semantically and syntactically. In other words, the case above results in a `ParseException` error solely because it contains two different SQL statements.

In essence, the `spark.sql()` method always expects a single SQL statement as input, and, therefore, it will try to parse this input query as a single SQL statement. If it finds multiple SQL statements inside this input string, the method will automatically raise the above error.

Now, be aware that some SQL queries can take multiple lines, but, **still be considered a single SQL statement**. A query started by a `WITH` clause is usually a good example of a SQL query that can group multiple `SELECT` statements, but still be considered a single SQL statement as a whole:

```
-- The query below would execute
-- perfectly fine inside spark.sql():
WITH table1 AS (
    SELECT *
```

```

        FROM somewhere
    ),

    filtering AS (
        SELECT *
        FROM table1
        WHERE dateOfTransaction == CAST("2022-02-02" AS DATE)
    )

    SELECT *
    FROM filtering

```

Another example of a usually big and complex query, that can take multiple lines but still be considered a single SQL statement, is a single SELECT statement that selects multiple subqueries that are nested together, like this:

```

-- The query below would also execute
-- perfectly fine inside spark.sql():
SELECT *
FROM (
    -- First subquery.
    SELECT *
    FROM (
        -- Second subquery..
        SELECT *
        FROM (
            -- Third subquery...
            SELECT *
            FROM (
                -- Ok this is enough....
            )
        )
    )
)

```

However, if we had multiple separate SELECT statements that were independent on each other, like in the example below, then, `spark.sql()` would issue an `ParseException` error if we tried to execute these three SELECT statements inside the same input string.

```

-- These three statements CANNOT be executed
-- at the same time inside spark.sql()
SELECT * FROM something;

```



```
SELECT * FROM somewhere;  
SELECT * FROM sometime;
```

As a conclusion, if you want to easily execute multiple statements, you can use a for loop which calls `spark.sql()` for each single SQL statement:

```
statements = '''SELECT * FROM something;  
SELECT * FROM somewhere;  
SELECT * FROM sometime;'''  
  
statements = statements.split('\n')  
for statement in statements:  
    spark.sql(statement)
```

## 7.2 Creating SQL Tables in Spark

In real life jobs at the industry, is very likely that your data will be allocated inside a SQL-like database. Spark can connect to a external SQL database through JDBC/ODBC connections, or, read tables from Apache Hive. This way, you can sent your SQL queries to this external database.

However, to expose more simplified examples throughout this chapter, we will use pyspark to create a simple temporary SQL table in our Spark SQL context, and use this temporary SQL table in our examples of SQL queries. This way, we avoid the work to connect to some existing SQL database, and, still get to learn how to use SQL queries in pyspark.

First, lets create our Spark Session. You can see below that I used the `config()` method to set a specific option of the session called `spark.sql.catalogImplementation`, to the value "hive". This option controls the implementation of the Spark SQL Catalog, which is a core part of the SQL functionality of Spark <sup>1</sup>.

Spark usually complain with a `AnalysisException` error when you try to create SQL tables with this option undefined (or not configured). So, if you decide to follow the examples of this chapter, please always set this option right at the start of your script <sup>2</sup>.

---

<sup>1</sup>There are some very good materials explaining what is the Spark SQL Catalog, and which is the purpose of it. For a soft introduction, I recommend Sarfaraz Hussain post: <https://medium.com/@sarfarazhussain211/metastore-in-apache-spark-9286097180a4>. For a more technical introduction, see <https://jaceklaskowski.gitbooks.io/mastering-spark-sql/content/spark-sql-Catalog.html>.

<sup>2</sup>You can learn more about why this specific option is necessary by looking at this StackOverflow post: <https://stackoverflow.com/questions/50914102/why-do-i-get-a-hive-support-is-required-to-create-hive-table-as-select-error>.

```
from pyspark.sql import SparkSession
spark = SparkSession\
    .builder\
    .config("spark.sql.catalogImplementation","hive")\
    .getOrCreate()
```

### 7.2.1 TABLEs versus VIEWs

To run a complete SQL query over any Spark DataFrame, you must register this DataFrame in the Spark SQL Catalog of your Spark Session. You can register a Spark DataFrame into this catalog as a physical SQL TABLE, or, as a SQL VIEW.

If you are familiar with the SQL language and Relational DataBase Management Systems - RDBMS (such as MySQL), you probably already heard of these two types (TABLE or VIEW) of SQL objects. But if not, we will explain each one in this section. It is worth pointing out that choosing between these two types **does not affect** your code, or your transformations in any way. It just affects the way that Spark SQL stores the table/DataFrame itself.

#### 7.2.1.1 VIEWs are stored as SQL queries or memory pointers

When you register a DataFrame as a SQL VIEW, the query to produce this DataFrame is stored, not the DataFrame itself. There are also cases where Spark stores a memory pointer instead, that points to the memory address where this DataFrame is stored in memory. In this perspective, Spark SQL uses this pointer every time it needs to access this DataFrame.

Therefore, when you call (or access) this SQL VIEW inside your SQL queries (for example, with a `SELECT * FROM` statement), Spark SQL will automatically get this SQL VIEW “on the fly” (or “on runtime”), by executing the query necessary to build the initial DataFrame that you stored inside this VIEW, or, if this DataFrame is already stored in memory, Spark will look at the specific memory address it is stored.

In other words, when you create a SQL VIEW, Spark SQL does not store any physical data or rows of the DataFrame. It just stores the SQL query necessary to build your DataFrame. In some way, you can interpret any SQL VIEW as an abbreviation to a SQL query, or a “nickname” to an already existing DataFrame.

As a consequence, for most “use case scenarios”, SQL VIEWs are easier to manage inside your data pipelines. Because you usually do not have to update them. Since they are calculated from scratch, at the moment you request for them, a SQL VIEW will always translate the most recent version of the data.

This means that the concept of a VIEW in Spark SQL is very similar to the concept of a VIEW in other types of SQL databases, such as the MySQL database. If you read the [official documentation for the CREATE VIEW statement at MySQL](#)<sup>3</sup> you will get a similar idea of a VIEW:

The `select_statement` is a SELECT statement that provides the definition of the view. (Selecting from the view selects, in effect, using the SELECT statement.) ...

The above statement, tells us that selecting a VIEW causes the SQL engine to execute the expression defined at `select_statement` using the SELECT statement. In other words, in MySQL, a SQL VIEW is basically an alias to an existing SELECT statement.

### 7.2.1.2 Differences in Spark SQL VIEWS

Although a Spark SQL VIEW being very similar to other types of SQL VIEW (such as the MySQL type), on Spark applications, SQL VIEWS are usually registered as TEMPORARY VIEWS<sup>4</sup> instead of standard (and “persistent”) SQL VIEW as in MySQL.

At MySQL there is no notion of a “temporary view”, although other popular kinds of SQL databases do have it, [such as the PostgreSQL database](#)<sup>5</sup>. So, a temporary view is not a exclusive concept of Spark SQL. However, is a special type of SQL VIEW that is not present in all popular kinds of SQL databases.

In other words, both Spark SQL and MySQL support the CREATE VIEW statement. In contrast, statements such as CREATE TEMPORARY VIEW and CREATE OR REPLACE TEMPORARY VIEW are available in Spark SQL, but not in MySQL.

### 7.2.1.3 Registering a Spark SQL VIEW in the Spark SQL Catalog

In pyspark, you can register a Spark DataFrame as a temporary SQL VIEW with the `createTempView()` or `createOrReplaceTempView()` DataFrame methods. These methods are equivalent to CREATE TEMPORARY VIEW and CREATE OR REPLACE TEMPORARY VIEW SQL statements of Spark SQL, respectively.

In essence, these methods register your Spark DataFrame as a temporary SQL VIEW, and have a single input, which is the name you want to give to this new SQL VIEW you are creating inside a string:

```
# To save the `df` DataFrame as a SQL VIEW, use one of the methods below:
df.createTempView('example_view')
df.createOrReplaceTempView('example_view')
```

---

<sup>3</sup><https://dev.mysql.com/doc/refman/8.0/en/create-view.html>

<sup>4</sup>I will explain more about the meaning of “temporary” at Section 7.2.2.

<sup>5</sup><https://www.postgresql.org/docs/current/sql-createview.html>

After we executed the above statements, we can now access and use the df DataFrame in any SQL query, like in the example below:

```
sql_query = '''
SELECT *
FROM example_view
WHERE value > 20
'''

spark.sql(sql_query).show()
```

```
+---+-----+-----+
| id|value|      date|
+---+-----+-----+
|  1|  28.3|2021-01-01|
|  3|  20.1|2021-01-02|
+---+-----+-----+
```

So you use the `createTempView()` or `createOrReplaceTempView()` methods when you want to make a Spark DataFrame created in pyspark (that is, a python object), available to Spark SQL.

Besides that, you also have the option to create a temporary VIEW by using pure SQL statements through the `sql()` method. However, when you create a temporary VIEW using pure SQL, you can only use (inside this VIEW) native SQL objects that are already stored inside your Spark SQL Context.

This means that you cannot make a Spark DataFrame created in python available to Spark SQL, by using a pure SQL inside the `sql()` method. To do this, you have to use the DataFrame methods `createTempView()` and `createOrReplaceTempView()`.

As an example, the query below uses pure SQL statements to create the `active_brazilian_users` temporary VIEW, which selects an existing SQL table called `hubspot.user_mails`:

```
CREATE TEMPORARY VIEW active_brazilian_users AS
SELECT *
FROM hubspot.user_mails
WHERE state == 'Active'
AND country_location == 'Brazil'
```

Temporary VIEWS like the one above (which are created from pure SQL statements being executed inside the `sql()` method) are kind of unusual in Spark SQL. Because you can easily avoid the work of creating a VIEW by using Common Table Expression (CTE) on a `WITH` statement, like in the query below:

```

WITH active_brazilian_users AS (
  SELECT *
  FROM hubspot.user_mails
  WHERE state == 'Active'
  AND country_location == 'Brazil'
)

SELECT A.user, SUM(sale_value), B.user_email
FROM sales.sales_per_user AS A
INNER JOIN active_brazilian_users AS B
GROUP BY A.user, B.user_email

```

Just as a another example, you can also run a SQL query that creates a persistent SQL VIEW (that is, without the TEMPORARY clause). In the example below, I am saving the simple query that I showed at the beginning of this chapter inside a VIEW called `list_of_codes`. This CREATE VIEW statement, register a persistent SQL VIEW in the SQL Catalog.

```

sql_query = '''
CREATE OR REPLACE VIEW list_of_codes AS
SELECT *
FROM (
  VALUES (11), (31), (24), (35)
) AS List(Codes)
'''

spark.sql(sql_query)

```

DataFrame[]

Now, every time I want to use this SQL query that selects a list of codes, I can use this `list_of_codes` as a shortcut:

```
spark.sql("SELECT * FROM list_of_codes").show()
```

```

+-----+
|Codes|
+-----+
|  11|
|  31|
|  24|
|  35|

```

+-----+

#### 7.2.1.4 TABLEs are stored as physical tables

In the other hand, SQL TABLEs are the “opposite” of SQL VIEWS. That is, SQL TABLEs are stored as physical tables inside the SQL database. In other words, each one of the rows of your table are stored inside the SQL database.

Because of this characteristic, when dealing with huges amounts of data, SQL TABLEs are usually faster to load and transform. Because you just have to read the data stored on the database, you do not need to calculate it from scratch every time you use it.

But, as a collateral effect, you usually have to physically update the data inside this TABLE, by using, for example, INSERT INTO statements. In other words, when dealing with SQL TABLE’s you usually need to create (and manage) data pipelines that are responsible for periodically update and append new data to this SQL TABLE, and this might be a big burden to your process.

#### 7.2.1.5 Registering Spark SQL TABLEs in the Spark SQL Catalog

In pyspark, you can register a Spark DataFrame as a SQL TABLE with the write.saveAsTable() DataFrame method. This method accepts, as first input, the name you want to give to this SQL TABLE inside a string.

```
# To save the `df` DataFrame as a SQL TABLE:  
df.write.saveAsTable('example_table')
```

As you expect, after we registered the DataFrame as a SQL table, we can now run any SQL query over example\_table, like in the example below:

```
spark.sql("SELECT SUM(value) FROM example_table").show()
```

```
+-----+  
|sum(value)|  
+-----+  
|      76.8|  
+-----+
```

You can also use pure SQL queries to create an empty SQL TABLE from scratch, and then, feed this table with data by using INSERT INTO statements. In the example below, we create a new database called examples, and, inside of it, a table called code\_brazil\_states. Then, we use multiple INSERT INTO statements to populate this table with few rows of data.

```

all_statements = '''CREATE DATABASE `examples`;
USE `examples`;
CREATE TABLE `code_brazil_states` (`code` INT, `state_name` STRING);
INSERT INTO `code_brazil_states` VALUES (31, "Minas Gerais");
INSERT INTO `code_brazil_states` VALUES (15, "Pará");
INSERT INTO `code_brazil_states` VALUES (41, "Paraná");
INSERT INTO `code_brazil_states` VALUES (25, "Paraíba");'''

statements = all_statements.split('\n')
for statement in statements:
    spark.sql(statement)

```

We can see now this new physical SQL table using a simple query like this:

```

spark\
    .sql('SELECT * FROM examples.code_brazil_states')\
    .show()

```

```

+----+-----+
|code| state_name|
+----+-----+
| 41|      Paraná|
| 31| Minas Gerais|
| 15|        Pará|
| 25|     Paraíba|
+----+-----+

```

#### 7.2.1.6 The different save “modes”

There are other arguments that you might want to use in the `write.saveAsTable()` method, like the `mode` argument. This argument controls how Spark will save your data into the database. By default, `write.saveAsTable()` uses the `mode = "error"` by default. In this mode, Spark will look if the table you referenced already exists, before it saves your data.

Let’s get back to the code we showed before (which is reproduced below). In this code, we asked Spark to save our data into a table called `"example_table"`. Spark will look if a table with this name already exists. If it does, then, Spark will raise an error that will stop the process (i.e. no data is saved).

```

df.write.saveAsTable('example_table')

```

Raising an error when you do not want to accidentally affect a SQL table that already exist, is a good practice. But, you might want to not raise an error in this situation. In case like this, you might want to

just ignore the operation, and get on with your life. For cases like this, `write.saveAsTable()` offers the `mode = "ignore"`.

So, in the code example below, we are trying to save the `df` DataFrame into a table called `example_table`. But if this `example_table` already exist, Spark will just silently ignore this operation, and will not save any data.

```
df.write.saveAsTable('example_table', mode = "ignore")
```

In addition, `write.saveAsTable()` offers two more different modes, which are `mode = "overwrite"` and `mode = "append"`. When you use one these two modes, Spark **will always save your data**, no matter if the SQL table you are trying to save into already exist or not. In essence, these two modes control whether Spark will delete or keep previous rows of the SQL table intact, before it saves any new data.

When you use `mode = "overwrite"`, Spark will automatically rewrite/replace the entire table with the current data of your DataFrame. In contrast, when you use `mode = "append"`, Spark will just append (or insert, or add) this data into the table. The subfigures at Figure 7.1 demonstrates these two modes visually.

You can see the full list of arguments of `write.SaveAsTable()`, and their description by [looking at the documentation](#)<sup>6</sup>.

## 7.2.2 Temporary versus Persistent sources

When you register any Spark DataFrame as a SQL TABLE, it becomes a persistent source. Because the contents, the data, the rows of the table are stored on disk, inside a database, and can be accessed any time, even after you close or restart your computer (or your Spark Session). In other words, it becomes “persistent” as in the sense of “it does not die”.

As another example, when you save a specific SQL query as a SQL VIEW with the `CREATE VIEW` statement, this SQL VIEW is saved inside the database. As a consequence, it becomes a persistent source as well, and can be accessed and reused in other Spark Sessions, unless you explicit drop (or “remove”) this SQL VIEW with a `DROP VIEW` statement.

However, with methods like `createTempView()` and `createOrReplaceTempView()` you register your Spark DataFrame as a *temporary* SQL VIEW. This means that the life (or time of existence) of this VIEW is tied to your Spark Session. In other words, it will exist in your Spark SQL Catalog only for the duration of your Spark Session. When you close your Spark Session, this VIEW just dies. When you start a new Spark Session it does not exist anymore. As a result, you have to register your DataFrame again at the catalog to use it one more time.

---

<sup>6</sup><https://spark.apache.org/docs/3.1.2/api/python/reference/api/pyspark.sql.DataFrameWriter.saveAsTable>



mode = "overwrite"



(a) Mode overwrite

mode = "append"



(b) Mode append

Figure 7.1: How Spark saves your data with different "save modes"

### 7.2.3 Spark SQL Catalog is the bridge between SQL and pyspark

Remember, to run SQL queries over any Spark DataFrame, you must register this DataFrame into the Spark SQL Catalog. Because of it, this Spark SQL Catalog works almost as the bridge that connects the python objects that hold your Spark DataFrames to the Spark SQL context. Without it, Spark SQL will not find your Spark DataFrames. As a result, it can not run any SQL query over it.

When you try to use a DataFrame that is not currently registered at the Spark SQL Catalog, Spark will automatically raise a `AnalysisException`, like in the example below:

```
spark\  
  .sql("SELECT * FROM this.does_not_exist")\  
  .show()
```

`AnalysisException: Table or view not found`

The methods `saveAsTable()`, `createTempView()` and `createOrReplaceTempView()` are the main methods to register your Spark DataFrame into this Spark SQL Catalog. This means that you have to use one of these methods before you run any SQL query over your Spark DataFrame.

## 7.3 The penguins DataFrame

Over the next examples in this chapter, we will explore the penguins DataFrame. This is the penguins dataset from the [palmerpenguins R library](https://allisonhorst.github.io/palmerpenguins/reference/penguins.html). It stores data of multiple measurements of penguin species from the islands in Palmer Archipelago.

These measurements include size (flipper length, body mass, bill dimensions) and sex, and they were collected by researchers of the Antarctica LTER program, a member of the Long Term Ecological Research Network. If you want to understand more about each field/column present in this dataset, I recommend you to read the [official documentation of this dataset](https://allisonhorst.github.io/palmerpenguins/reference/penguins.html)<sup>7</sup>.

To get this data, you can download the CSV file called `penguins.csv` (remember that this CSV can be downloaded from the book repository<sup>8</sup>). In the code below, I am reading this CSV file and creating a Spark DataFrame with its data. Then, I register this Spark DataFrame as a SQL temporary view (called `penguins_view`) using the `createOrReplaceTempView()` method.

```
path = "../Data/penguins.csv"  
penguins = spark.read\  
  .csv(path, header = True)
```

---

<sup>7</sup><https://allisonhorst.github.io/palmerpenguins/reference/penguins.html>

<sup>8</sup><https://github.com/pedropark99/Introd-pyspark/tree/main/Data>

```
penguins.createOrReplaceTempView('penguins_view')
```

After these commands, I have now a SQL view called `penguins_view` registered in my Spark SQL context, which I can query it, using pure SQL:

```
spark.sql('SELECT * FROM penguins_view').show(5)
```

```
+-----+-----+-----+-----+-----+
|species|  island|bill_length_mm|bill_depth_mm|flipper_length_mm|body_mass_g|
+-----+-----+-----+-----+-----+
| Adelie|Torgersen|          39.1|          18.7|           181|       3750|
| Adelie|Torgersen|          39.5|          17.4|           186|       3800|
| Adelie|Torgersen|          40.3|           18|           195|       3250|
| Adelie|Torgersen|          NULL|          NULL|           NULL|          NULL|
| Adelie|Torgersen|          36.7|          19.3|           193|       3450|
+-----+-----+-----+-----+-----+
```

only showing top 5 rows

... with 2 more columns: sex, year

## 7.4 Selecting your Spark DataFrames

An obvious way to access any SQL TABLE or VIEW registered in your Spark SQL context, is to select it, through a simple `SELECT * FROM` statement, like we saw in the previous examples. However, it can be quite annoying to type “`SELECT * FROM`” every time you want to use a SQL TABLE or VIEW in Spark SQL.

That is why Spark offers a shortcut to us, which is the `table()` method of your Spark session. In other words, the code `spark.table("table_name")` is a shortcut to `spark.sql("SELECT * FROM table_name")`. They both mean the same thing. For example, we could access `penguins_view` as:

```
spark\
  .table('penguins_view')\
  .show(5)
```

```
+-----+-----+-----+-----+-----+
|species|  island|bill_length_mm|bill_depth_mm|flipper_length_mm|body_mass_g|
+-----+-----+-----+-----+-----+
| Adelie|Torgersen|          39.1|          18.7|           181|       3750|
| Adelie|Torgersen|          39.5|          17.4|           186|       3800|
```

Adelie Torgersen	40.3	18	195	3250
Adelie Torgersen	NULL	NULL	NULL	NULL
Adelie Torgersen	36.7	19.3	193	3450

+-----+-----+-----+-----+-----+-----+-----+-----+  
only showing top 5 rows  
... with 2 more columns: sex, year

## 7.5 Executing SQL expressions

As I noted at Section 4.2, columns of a Spark DataFrame (or objects of class Column) are closely related to expressions. As a result, you usually use and execute expressions in Spark when you want to transform (or mutate) columns of a Spark DataFrame.

This is no different for SQL expressions. A SQL expression is basically any expression you would use on the SELECT statement of your SQL query. As you can probably guess, since they are used in the SELECT statement, these expressions are used to transform columns of a Spark DataFrame.

There are many column transformations that are particularly verbose and expensive to write in “pure” pyspark. But you can use a SQL expression in your favor, to translate this transformation into a more short and concise form. Virtually any expression you write in pyspark can be translated into a SQL expression.

To execute a SQL expression, you give this expression inside a string to the `expr()` function from the `pyspark.sql.functions` module. Since expressions are used to transform columns, you normally use the `expr()` function inside a `withColumn()` or a `select()` DataFrame method, like in the example below:

```
from pyspark.sql.functions import expr

spark\
  .table('penguins_view')\
  .withColumn(
    'specie_island',
    expr("CONCAT(species, '_', island)")
  )\
  .withColumn(
    'sex_short',
    expr("CASE WHEN sex == 'male' THEN 'M' ELSE 'F' END")
  )\
  .select('specie_island', 'sex_short')\
  .show(5)
```

```

+-----+-----+
|  specie_island|sex_short|
+-----+-----+
|Adelie_Torgersen|      M|
|Adelie_Torgersen|      F|
|Adelie_Torgersen|      F|
|Adelie_Torgersen|      F|
|Adelie_Torgersen|      F|
+-----+-----+
only showing top 5 rows

```

I particularly like to write “if-else” or “case-when” statements using a pure CASE WHEN SQL statement inside the `expr()` function. By using this strategy you usually get a more simple statement that translates the intention of your code in a cleaner way. But if I wrote the exact same CASE WHEN statement above using pure pyspark functions, I would end up with a shorter (but “less clean”) statement using the `when()` and `otherwise()` functions:

```

from pyspark.sql.functions import (
    when, col,
    concat, lit
)

spark\
    .table('penguins_view')\
    .withColumn(
        'specie_island',
        concat('species', lit('_'), 'island')
    )\
    .withColumn(
        'sex_short',
        when(col("sex") == 'male', 'M')\
            .otherwise('F')
    )\
    .select('specie_island', 'sex_short')\
    .show(5)

```

```

+-----+-----+
|  specie_island|sex_short|
+-----+-----+
|Adelie_Torgersen|      M|
|Adelie_Torgersen|      F|
|Adelie_Torgersen|      F|

```

```
|Adelie_Torgersen|      F|
|Adelie_Torgersen|      F|
+-----+-----+
only showing top 5 rows
```

## 7.6 Every DataFrame transformation in Python can be translated into SQL

All DataFrame API transformations that you write in Python (using pyspark) can be translated into SQL queries/expressions using the Spark SQL module. Since the DataFrame API is a core part of pyspark, the majority of python code you write with pyspark can be translated into SQL queries (if you want to).

Is worth pointing out, that, no matter which language you choose (Python or SQL), they are both further compiled to the same base instructions. The end result is that the Python code you write and his SQL translated version **will perform the same** (they have the same efficiency), because they are compiled to the same instructions before being executed by Spark.

### 7.6.1 DataFrame methods are usually translated into SQL keywords

When you translate the methods from the python DataFrame class (like `orderBy()`, `select()` and `where()`) into their equivalents in Spark SQL, you usually get SQL keywords (like `ORDER BY`, `SELECT` and `WHERE`).

For example, if I needed to get the top 5 penguins with the biggest body mass at `penguins_view`, that had sex equal to "female", and, ordered by bill length, I could run the following python code:

```
from pyspark.sql.functions import col
top_5 = penguins\
    .where(col('sex') == 'female')\
    .orderBy(col('body_mass_g').desc())\
    .limit(5)

top_5\
    .orderBy('bill_length_mm')\
    .show()
```

```
+-----+-----+-----+-----+-----+-----+
|species|island|bill_length_mm|bill_depth_mm|flipper_length_mm|body_mass_g|
+-----+-----+-----+-----+-----+-----+
|Adelie_
```

Gentoo Biscoe	44.9	13.3	213	5100
Gentoo Biscoe	45.1	14.5	207	5050
Gentoo Biscoe	45.2	14.8	212	5200
Gentoo Biscoe	46.5	14.8	217	5200
Gentoo Biscoe	49.1	14.8	220	5150

+-----+-----+-----+-----+-----+-----+-----+-----+  
... with 2 more columns: sex, year

I could translate the above python code to the following SQL query:

```
WITH top_5 AS (
    SELECT *
    FROM penguins_view
    WHERE sex == 'female'
    ORDER BY body_mass_g DESC
    LIMIT 5
)

SELECT *
FROM top_5
ORDER BY bill_length_mm
```

Again, to execute the above SQL query inside pyspark we need to give this query as a string to the `sql()` method of our Spark Session, like this:

```
query = '''
WITH top_5 AS (
    SELECT *
    FROM penguins_view
    WHERE sex == 'female'
    ORDER BY body_mass_g DESC
    LIMIT 5
)

SELECT *
FROM top_5
ORDER BY bill_length_mm
'''

# The same result of the example above
spark.sql(query).show()
```

```
+-----+-----+-----+-----+-----+-----+
|species|island|bill_length_mm|bill_depth_mm|flipper_length_mm|body_mass_g|
+-----+-----+-----+-----+-----+-----+
| Gentoo|Biscoe|         44.9|         13.3|          213|       5100|
| Gentoo|Biscoe|         45.1|         14.5|          207|       5050|
| Gentoo|Biscoe|         45.2|         14.8|          212|       5200|
| Gentoo|Biscoe|         46.5|         14.8|          217|       5200|
| Gentoo|Biscoe|         49.1|         14.8|          220|       5150|
+-----+-----+-----+-----+-----+-----+
... with 2 more columns: sex, year
```

## 7.6.2 Spark functions are usually translated into SQL functions

Every function from the `pyspark.sql.functions` module you might use to describe your transformations in python, can be directly used in Spark SQL. In other words, every Spark function that is accesible in python, is also accesible in Spark SQL.

When you translate these python functions into SQL, they usually become a pure SQL function with the same name. For example, if I wanted to use the `regexp_extract()` python function, from the `pyspark.sql.functions` module in Spark SQL, I just use the `REGEXP_EXTRACT()` SQL function. The same occurs to any other function, like the `to_date()` function for example.

```
from pyspark.sql.functions import to_date, regexp_extract
# `df1` and `df2` are both equal. Because they both
# use the same `to_date()` and `regexp_extract()` functions
df1 = (spark
      .table('penguins_view')
      .withColumn(
        'extract_number',
        regexp_extract('bill_length_mm', '[0-9]+', 0)
      )
      .withColumn('date', to_date('year', 'y'))
      .select(
        'bill_length_mm', 'year',
        'extract_number', 'date'
      )
    )

df2 = (spark
      .table('penguins_view')
      .withColumn(
        'extract_number',
```



```

    expr("REGEXP_EXTRACT(bill_length_mm, '[0-9]+', 0)")
  )
  .withColumn('date', expr("TO_DATE(year, 'y')"))
  .select(
    'bill_length_mm', 'year',
    'extract_number', 'date'
  )
)

df2.show(5)

```

```

+-----+-----+-----+-----+
|bill_length_mm|year|extract_number|    date|
+-----+-----+-----+-----+
|          39.1|2007|          39|2007-01-01|
|          39.5|2007|          39|2007-01-01|
|          40.3|2007|          40|2007-01-01|
|          NULL|2007|         NULL|2007-01-01|
|          36.7|2007|          36|2007-01-01|
+-----+-----+-----+-----+

```

only showing top 5 rows

This is very handy. Because for every new python function from the `pyspark.sql.functions` module, that you learn how to use, you automatically learn how to use in Spark SQL as well, because is the same function, with the basically the same name and arguments.

As an example, I could easily translate the above transformations that use the `to_date()` and `regexp_extract()` python functions, into the following SQL query (that I could easily execute trough the `sql()` Spark Session method):

```

SELECT
  bill_length_mm, year,
  REGEXP_EXTRACT(bill_length_mm, '[0-9]+', 0) AS extract_number,
  TO_DATE(year, 'y') AS date
FROM penguins_view

```

## 8 Transforming your Spark DataFrame - Part 2

At Chapter 5 I introduced six core types of transformations over Spark DataFrames. In this chapter, I will expand your knowledge by introducing six more common types of transformations available to Spark DataFrames, which are:

- Replacing null values;
- Removing duplicated values;
- Merging multiple DataFrames with UNION operations;
- Merging multiple DataFrames with JOIN operations;
- Rows to columns and columns to rows with Pivot operations;
- Collecting and explode operations;

### 8.1 Removing duplicated values from your DataFrame

Removing duplicated values from DataFrames is a very common operation in ETL pipelines. In pyspark you have two options to remove duplicated values, which are:

- `distinct()` which removes all duplicated values considering the combination of all current columns in the DataFrame;
- `drop_duplicates()` or `dropDuplicates()` which removes all duplicated values considering a specific combination (or set) of columns in the DataFrame;

These three methods above are all DataFrames methods. Furthermore, the methods `drop_duplicates()` and `dropDuplicates()` are equivalent. They both mean the same thing, and have the same arguments and perform the same operation.

When you run `drop_duplicates()` or `dropDuplicates()` without any argument, they automatically use by default the combination of all columns available in the DataFrame to identify the duplicated values. As a consequence, over this specific situation, the methods `drop_duplicates()` or `dropDuplicates()` become equivalent to the `distinct()` method. Because they use the combination of all columns in the DataFrame.

Lets pick the `supermarket_sales` DataFrame exposed below as an example. You can see below, that this DataFrame contains some duplicated values, specifically on the transaction IDs “T001” e “T004”. We also have some “degree of duplication” on the transaction ID “T006”. But the two rows describing this ID “T006” are not precisely identical, since they have a small difference on the `quantity` column.

```

from pyspark.sql.types import (
    StructType, StructField,
    StringType, IntegerType,
    FloatType
)

schema = StructType([
    StructField("transaction_id", StringType(), True),
    StructField("product_name", StringType(), True),
    StructField("quantity", IntegerType(), True),
    StructField("price", FloatType(), True)
])

data = [
    ("T001", "Apple", 5, 1.2),
    ("T001", "Apple", 5, 1.2),
    ("T002", "Banana", 3, 0.8),
    ("T004", "Mango", 2, 2.0),
    ("T004", "Mango", 2, 2.0),
    ("T004", "Mango", 2, 2.0),
    ("T005", "Grapes", 1, 3.5),
    ("T006", "Apple", 2, 1.2),
    ("T006", "Apple", 1, 1.2),
    ("T007", "Banana", 4, 0.8),
    ("T008", "Apple", 3, 1.2)
]

supermarket_sales = spark.createDataFrame(data, schema)
supermarket_sales.show(6)

```

```

+-----+-----+-----+-----+
|transaction_id|product_name|quantity|price|
+-----+-----+-----+-----+
|          T001|        Apple|         5|    1.2|
|          T001|        Apple|         5|    1.2|
|          T002|       Banana|         3|    0.8|
|          T004|        Mango|         2|    2.0|
|          T004|        Mango|         2|    2.0|
|          T004|        Mango|         2|    2.0|
+-----+-----+-----+-----+

```

only showing top 6 rows

We can remove these duplicated values by using the `distinct()` method. In the example of transaction ID “T004”, all duplicated rows of this ID contains the same values (“T004”, “Mango”, 2, 2.0), precisely in this order. Because of that, the `distinct()` method is enough to remove all of these duplicated values from the table.

```
supermarket_sales\  
  .distinct()\  
  .show(6)
```

```
+-----+-----+-----+-----+  
|transaction_id|product_name|quantity|price|  
+-----+-----+-----+-----+  
|          T001|        Apple|         5|  1.2|  
|          T002|        Banana|         3|  0.8|  
|          T004|         Mango|         2|  2.0|  
|          T005|         Grapes|         1|  3.5|  
|          T006|         Apple|         2|  1.2|  
|          T006|         Apple|         1|  1.2|  
+-----+-----+-----+-----+
```

only showing top 6 rows

However, the two rows describing the transaction ID “T006” have some difference on the quantity column, and as a result, the `distinct()` method does not identify these two rows as “duplicated values”, and they are not removed from the input DataFrame.

Now, if we needed a DataFrame that contained one row for each transaction ID (that is, the values on `transaction_id` column must be unique), we could use the `drop_duplicates()` method with only the column `transaction_id` as the key to remove all duplicated values of this column. This way, we get a slightly different output as you can see below.

```
supermarket_sales\  
  .drop_duplicates(['transaction_id'])\  
  .show(8)
```

```
+-----+-----+-----+-----+  
|transaction_id|product_name|quantity|price|  
+-----+-----+-----+-----+  
|          T001|        Apple|         5|  1.2|  
|          T002|        Banana|         3|  0.8|  
|          T004|         Mango|         2|  2.0|  
|          T005|         Grapes|         1|  3.5|  
|          T006|         Apple|         2|  1.2|
```

	T007	Banana	4	0.8
	T008	Apple	3	1.2
+-----+-----+-----+-----+				

In the example above, the duplicated values of IDs “T001” and “T004” were removed as we expected. But we also removed the second value for ID “T006”. Because we did not listed the quantity column on `drop_duplicates()`, and, as a result, the `drop_duplicates()` method was not concerned with the differences on the quantity column. In other words, it used solely the `transaction_id` column to identify the duplicated values.

## 8.2 Other techniques for dealing with null values

At Section 5.5.5 I showed how you can use `filter()` or `where()` DataFrame methods to remove all rows that contained a null value on some column. There are two other DataFrames methods available in Spark that you might want use to deal with null values. In essence, you can either remove or replace these null values.

### 8.2.1 Replacing null values

Instead of removing the null values, and pretending that they never existed, maybe, you prefer to replace these null values by a more useful or representative value, such as 0, or an empty string (' '), or a False value, etc. To do that in pyspark, we can use the `na.fill()` and `fillna()` DataFrame methods.

Both methods mean the same thing, and they work the exact same way. The most popular way of using this methods, is to provide a python dict as input. Inside this dict you have key-value pairs, where the key represents the column name, and the value represents the static value that will replace all null values that are found on the column specified by the key.

In the example below, I created a simple `df` DataFrame which contains some null values on the age column. By providing the dict `{'age': 0}` to `fillna()`, I am asking `fillna()` to replace all null values found on the age column by the value 0 (zero).

```
data = [
    (1, "John", None, "2023-04-05"),
    (2, "Alice", 25, "2023-04-09"),
    (3, "Bob", None, "2023-04-12"),
    (4, "Jane", 30, None),
    (5, "Mike", 35, None)
]
columns = ["id", "name", "age", "date"]
df = spark.createDataFrame(data, columns)
```

```
# Or `df.na.fill({'age': 0}).show()`
# It is the same thing
df.fillna({'age': 0}).show()
```

```
+---+-----+---+-----+
| id| name|age|      date|
+---+-----+---+-----+
|  1| John|  0|2023-04-05|
|  2|Alice| 25|2023-04-09|
|  3|  Bob|  0|2023-04-12|
|  4| Jane| 30|      NULL|
|  5| Mike| 35|      NULL|
+---+-----+---+-----+
```

You can see in the above example, that the null values present in the date column were maintained intact on the result. Because we did not ask to `fillna()` to replace the values of this column, by including it on the input dict that we provided.

If we do include this date column on the input dict, then, `fillna()` will take care of this column as well:

```
df.fillna({'age': 0, 'date': '2023-01-01'})\
.show()
```

```
+---+-----+---+-----+
| id| name|age|      date|
+---+-----+---+-----+
|  1| John|  0|2023-04-05|
|  2|Alice| 25|2023-04-09|
|  3|  Bob|  0|2023-04-12|
|  4| Jane| 30|2023-01-01|
|  5| Mike| 35|2023-01-01|
+---+-----+---+-----+
```

### 8.2.2 Dropping all null values

Spark also offers the `na.drop()` and `dropna()` DataFrames methods, which you can use to easily remove any row that contains a null value on any column of the DataFrame. This is different from `filter()` and `where()`, because on these two methods you have to build a logical expression that translate “not-null values”.

In contrast, on `na.drop()` and `dropna()` methods you do not have a logical expression. You just call these methods, and they do the heavy work for you. They search through the entire DataFrame. When it identifies a null value on the DataFrame, it removes the entire row that contains such null value.

For example, if we apply these methods on the `df` DataFrame that we used on the previous section, this is the end result:

```
df.na.drop()\n.show()
```

```
+---+-----+---+-----+
| id| name|age|      date|
+---+-----+---+-----+
|  2|Alice| 25|2023-04-09|
+---+-----+---+-----+
```

## 8.3 Union operations

When you have many individual DataFrames that have the same columns, and you want to unify them into a single big DataFrame that have all the rows from these different DataFrames, you want to perform an UNION operation.

An UNION operation works on a pair of DataFrames. It returns the row-wise union of these two DataFrames. In pyspark, we perform UNION operations by using the `union()` DataFrame method. To use this method, you just provide the other DataFrame you want to make the union with. So the expression `df1.union(df2)` creates a new DataFrame which contains all the rows from both the `df1` and `df2` DataFrames.

Moreover, in common SQL engines there are usually two kinds of UNION operations, which are: *union all* and *union distinct*. When you use an *union all* operation, you are saying that you just want to unify the two DataFrames, no matter what data you find in each one of them. You do not care if duplicated values are generated in the process, because an observation “x” might be present both on `df1` and `df2`.

In contrast, an *union distinct* operation is the exact opposite of that. It merges the rows from both DataFrames together, and then, it removes all duplicated values from the result. So you use an *union distinct* operation when you want a single DataFrame that contains all rows from both DataFrames `df1` and `df2`, but, you do not want any duplicated rows into this single DataFrame.

By default, the `union()` method always performs an *union all* operation. However, to do an *union distinct* operation in pyspark, you actually have to use the `union()` method in conjunction with the `distinct()` or `drop_duplicates()` methods. In other words, there is not a direct method in pyspark that performs an *union distinct* operation on a single command.

Look at the example below with df1 and df2 DataFrames.

```
df1 = [  
  (1, 'Anne', 'F'),  
  (5, 'Mike', 'M'),  
  (2, 'Francis', 'M'),  
]  
  
df2 = [  
  (5, 'Mike', 'M'),  
  (7, 'Arthur', 'M'),  
  (1, 'Anne', 'F'),  
]  
  
df1 = spark.createDataFrame(df1, ['ID', 'Name', 'Sex'])  
df2 = spark.createDataFrame(df2, ['ID', 'Name', 'Sex'])  
  
# An example of UNION ALL operation:  
df1.union(df2).show()
```

```
+---+-----+---+  
| ID|   Name|Sex|  
+---+-----+---+  
|  1|   Anne| F|  
|  5|   Mike| M|  
|  2|Francis| M|  
|  5|   Mike| M|  
|  7| Arthur| M|  
|  1|   Anne| F|  
+---+-----+---+
```

```
# An example of UNION DISTINCT operation  
df1\  
  .union(df2)\  
  .distinct\  
  .show()
```

```
+---+-----+---+  
| ID|   Name|Sex|  
+---+-----+---+  
|  1|   Anne| F|
```



```
| 5|   Mike|  M|
| 2|Francis|  M|
| 7| Arthur|  M|
+---+-----+---+
```

Because an UNION operation merges the two DataFrames in a vertical way, the columns between the two DataFrames must match. If the columns between the two DataFrames are not in the same places, a mismatch happens during the operation, and Spark will do nothing to fix your mistake.

Most programming languages would issue an error at this point, warning you about this conflict between the columns found on each DataFrame and their respective positions. However, in Spark, if the columns are out of order, Spark will continue with the UNION operation, as if nothing was wrong. Spark will not even raise a warning for you. Since this problem can easily pass unnoticed, be aware of it.

In the example below, we have a third DataFrame called df3. Notice that the columns in df3 are the same of df1 and df2. However, the columns from df3 are in a different order than in df1 and df2.

```
data = [
    ('Marla', 'F', 9),
    ('Andrew', 'M', 15),
    ('Peter', 'M', 12)
]
df3 = spark.createDataFrame(data, ['Name', 'Sex', 'ID'])
df3.show()
```

```
+-----+-----+---+
|  Name|Sex| ID|
+-----+-----+---+
| Marla|  F|  9|
|Andrew|  M| 15|
| Peter|  M| 12|
+-----+-----+---+
```

If we try to perform an UNION operation between, let's say, df2 and df3, the operations just works. But, the end result of this operation is not correct, as you can see in the example below.

```
df2.union(df3).show()
```

```
+-----+-----+---+
|    ID|  Name|Sex|
+-----+-----+---+
|     5|  Mike|  M|
```

```

|      7|Arthur| M|
|      1|  Anne| F|
| Marla|      F| 9|
|Andrew|      M|15|
| Peter|      M|12|
+-----+-----+-----+

```

Although this might be problematic, Spark provides an easy-to-use solution when the columns are in different places between each DataFrame. This solution is the `unionByName()` method.

The difference between `union()` and `unionByName()` methods, is that the `unionByName()` method makes an matching by column name, before it performs the UNION. In other words, it compares the column names found on each DataFrame and it matches each column by its name. This way, the columns present on each DataFrame of the UNION must have the same name, but they do not need to be in the same positions on both DataFrames.

If we use this method on the same example as above, you can see below that we get a different result, and a correct one this time.

```

df2.unionByName(df3)\
    .show()

```

```

+---+-----+-----+
| ID|  Name|Sex|
+---+-----+-----+
| 5|  Mike| M|
| 7|Arthur| M|
| 1|  Anne| F|
| 9| Marla| F|
|15|Andrew| M|
|12| Peter| M|
+---+-----+-----+

```

Therefore, if you want to make an UNION operation between two DataFrames, you can generally use the `union()` method. But if you suspect the columns from these DataFrames might be in different positions on each DataFrame, you can change to the `unionByName()` method.

In contrast, if the columns are different not only on position, but also, on column name, then, `unionByName()` will not work. The two DataFrames involved on an UNION operation must be very similar. If they are not similar, then, you will have a hard time trying to do the operation.

Another problem that you might face is if you try to unify two DataFrames that have different numbers of columns between them. In this situation, it means that the two DataFrames have “different widths”,

and, as a result of that, an `AnalysisException` error will be raised by Spark if you try to unify them with an `UNION` operation, like in the example below:

```
from pyspark.sql.types import (
    StructField,
    StructType,
    LongType
)

schema = StructType([StructField('ID', LongType(), False)])
df4 = [
    (19,), (17,), (16,)
]
df4 = spark.createDataFrame(df4, schema)
df3.union(df4).show()
```

```
AnalysisException: Union can only be performed on tables
with the same number of columns, but the first table has
3 columns and the second table has 1 columns;
'Union false, false
:- LogicalRDD [Name#703, Sex#704, ID#705L], false
+- LogicalRDD [ID#762L], false
```

## 8.4 Join operations

A `JOIN` operation is another very common operation that is also used to bring data from scattered sources into a single unified `DataFrame`. In `pyspark`, we can build `JOIN` operations by using the `join()` `DataFrame` method. This method accepts three arguments, which are:

- `other`: the `DataFrame` you want to `JOIN` with (i.e. the `DataFrame` on the right side of the `JOIN`);
- `on`: a column name, or a list of column names, that represents the key (or keys) of the `JOIN`;
- `how`: the kind of `JOIN` you want to perform (inner, full, left, right);

As a first example, let's use the `info` and `band_instruments` `DataFrames`. With the source code below, you can quickly re-create these two `DataFrames` in your session:

```
info = [
    ('Mick', 'Rolling Stones', '1943-07-26', True),
    ('John', 'Beatles', '1940-09-10', True),
    ('Paul', 'Beatles', '1942-06-18', True),
    ('George', 'Beatles', '1943-02-25', True),
```

```

    ('Ringo', 'Beatles', '1940-07-07', True)
]

info = spark.createDataFrame(
    info,
    ['name', 'band', 'born', 'children']
)

band_instruments = [
    ('John', 'guitar'),
    ('Paul', 'bass'),
    ('Keith', 'guitar')
]

band_instruments = spark.createDataFrame(
    band_instruments,
    ['name', 'plays']
)

```

If you look closely to these two DataFrames, you will probably notice that they both describe musicians from two famous rock bands from 60's and 70's. The info DataFrame have more personal or general informations about the musicians, while the band\_instruments DataFrame have only data about the main musical instruments that they play.

```
info.show()
```

```

+-----+-----+-----+-----+
| name|      band|    born|children|
+-----+-----+-----+-----+
| Mick|Rolling Stones|1943-07-26|    true|
| John|      Beatles|1940-09-10|    true|
| Paul|      Beatles|1942-06-18|    true|
|George|      Beatles|1943-02-25|    true|
| Ringo|      Beatles|1940-07-07|    true|
+-----+-----+-----+-----+

```

```
band_instruments.show()
```

```

+-----+-----+
| name| plays|

```

```
+-----+-----+
| John|guitar|
| Paul| bass|
|Keith|guitar|
+-----+-----+
```

It might be of your interest, to have a single DataFrame that contains both the personal information and the musical instrument of each musician. In this case, you can build a JOIN operation between these DataFrames to get this result. An example of this JOIN in pyspark would be:

```
info.join(band_instruments, on = 'name', how = 'left')\
    .show(5)
```

```
+-----+-----+-----+-----+-----+
| name|          band|      born|children| plays|
+-----+-----+-----+-----+-----+
| Mick|Rolling Stones|1943-07-26|    true|  NULL|
| John|      Beatles|1940-09-10|    true|guitar|
| Paul|      Beatles|1942-06-18|    true|  bass|
|George|      Beatles|1943-02-25|    true|  NULL|
| Ringo|      Beatles|1940-07-07|    true|  NULL|
+-----+-----+-----+-----+-----+
```

In the example above, we are performing a *left join* between the two DataFrames, using the name column as the JOIN key. Now, we have a single DataFrame with all 5 columns from both DataFrames (plays, children, name, band and born).

### 8.4.1 What is a JOIN ?

I imagine you are already familiar with JOIN operations. However, in order to build good and precise JOIN operations, is very important to know what a JOIN operation actually is. So let's revisit it.

A JOIN operation merges two different DataFrames together into a single unified DataFrame. It does this by using a column (or a set of columns) as keys to identify the observations of both DataFrames, and connects these observations together.

A JOIN (like UNION) is also an operation that works on a pair of DataFrames. It is very common to refer to this pair as “the sides of the JOIN”. That is, the DataFrame on the left side of the JOIN, and the DataFrame on the right side of the JOIN. Or also, the DataFrames “A” (left side) and “B” (right side).

The main idea (or objective) of the JOIN is to bring all data from the DataFrame on the right side, into the DataFrame on the left side. In other words, a JOIN between DataFrames A and B results into a DataFrame C which contains all columns and rows from both DataFrames A and B.

In an UNION operation, both DataFrames must have the same columns, because in an UNION operation you are concatenating both DataFrames together vertically, so the number of columns (or the “width” of the tables) need to match. However, in a JOIN operation, both DataFrames only need to have at least one column in common. Apart from that, in a JOIN, both DataFrames can have very different structure and columns from each other.

One key characteristic of JOIN operations is its key matching mechanism. A JOIN uses the columns you provide to **build a key**. This key is used to identify rows (or “observations”) in both DataFrames. In other words, these keys identify relationships between the two DataFrames. These relations are vital to the JOIN.

If we go back to `info` and `band_instruments` DataFrames, and analyse them for a bit more, we can see that they both have a `name` column which contains the name of the musician being described on the current row. This `name` column can be used as **the key** of the JOIN. Because this column is available on both DataFrames, and it can be used to identify a single observation (or a single musician) present in each DataFrame.

So the JOIN key is a column (or a combination of columns) that can identify what observations are (and are not) present on both DataFrames. At Figure 8.1, we can see the observations from `info` and `band_instruments` in a visual manner. You see in the figure that both Paul and John are described in both DataFrames. At the same time, Ringo, Mick and George are present only on `info`, while Keith is only at `band_instruments`.

In a certain way, you can see the JOIN key as a way to **identify relationships between the two DataFrames**. A JOIN operation uses these relationships to merge your DataFrames in a precise way. A JOIN does not simply horizontally glue two DataFrames together. It uses the JOIN key to perform a matching process between the observations of the two DataFrames.

This matching process ensures that the data present in DataFrame “B” is correctly transported to the DataFrame “A”. In other words, it ensures that the oranges are paired with oranges, apples with apples, bananas with bananas, you got it.

Just to describe visually what this matching process is, we have the Figure 8.2 below. In this figure, we have two DataFrames on the left and center of the image, which represents the inputs of the JOIN. We also have a third DataFrame on the right side of the image, which is the output (or the result) of the JOIN.

In this specific example, the column that represents the JOIN key is the `ID` column. Not only this column is present on both DataFrames, but it also represents a unique identifier to each person described in both tables. And that is precisely the job of a JOIN key. It represents a way to identify observations (or “persons”, or “objects”, etc.) on both tables.

<span style="display: inline-block; width: 20px; height: 10px; background-color: #008080; border: 1px solid black;"></span> Musician present only at DataFrame <b>info</b>	<span style="display: inline-block; width: 20px; height: 10px; background-color: #ff0000; border: 1px solid black;"></span> Musician present at both DataFrames	<span style="display: inline-block; width: 20px; height: 10px; background-color: #ffa500; border: 1px solid black;"></span> Musician present only at DataFrame <b>band_instruments</b>
--	---	--

DataFrame <b>info</b>			
name	band	born	children
Mick	Rolling Stones	1943-07-26	TRUE
John	Beatles	1940-09-10	TRUE
Paul	Beatles	1942-06-18	TRUE
George	Beatles	1943-02-25	TRUE
Ringo	Beatles	1940-07-07	TRUE

DataFrame <b>band_instruments</b>	
name	plays
John	guitar
Paul	bass
Keith	guitar

Figure 8.1: The relations between info and band\_instruments DataFrames

You can see at Figure 8.2, that when the ID 100 is found on the 1st row of the left DataFrame, the JOIN initiates a lookup/matching process on the center DataFrame, looking for a row in the DataFrame that matches this ID 100. When it finds this ID 100 (on the 4th row of the center DataFrame), it captures and connects these two rows on both DataFrames, because these rows describes the same person (or observation), and because of that, they should be connected. This same matching process happens for all remaining ID values.

## 8.4.2 The different types of JOIN

JOIN operations actually comes in different flavours (or types). The four main known types of JOINS are: *full*, *left*, *right* and *inner*. All of these different types of JOIN perform the same steps and matching processes that we described on the previous section. But they differ on the treatment they do to unmatched observations. In other words, these different types of JOINS differ on **what they do in cases when an observation is not found on both DataFrames of the JOIN** (e.g. when an observation is found only on table A).

In other words, all these four types will perform the same matching process between the two DataFrames, and will connect observations that are found in both DataFrames. However, which rows are included in the final output is what changes between each type (or “flavour”) of JOIN.

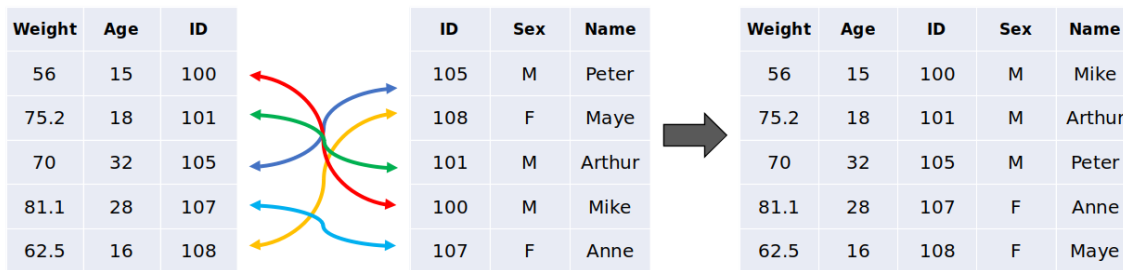


Figure 8.2: The matching process of a JOIN operation

In this situation, the words “left” and “right” are identifiers to the DataFrames involved on the JOIN operation. That is, the word *left* refers to the DataFrame on the left side of the JOIN, while the word *right* refers to the DataFrame on the right side of the JOIN.

A very useful way of understanding these different types of JOINS is to represent both DataFrames as numerical sets (as we learn in mathematics). The Figure 8.3 gives you a visual representation of each type of JOIN using this “set model” of representing JOINS. Remember, all of these different types of JOIN work the same way, they just do different actions when an observation is not found on both tables.

The most “complete” and “greedy” type of JOIN is the *full join*. Because this type returns all possible combinations of both DataFrames. In other words, this type of JOIN will result in a DataFrame that have all observations from both DataFrames. It does not matter if an observation is present only on table A, or only on table B, or maybe, on both tables. A *full join* will always try to connect as much observation as it can.

That is why the *full join* is represented on Figure 8.3 as the union between the two tables (or the two sets). In contrast, an *inner join* is the intersection of the two tables (or two sets). That is, an *inner join* will result in a new DataFrame which contains solely the observations that could be found on both tables. If a specific observation is found only on one table of the JOIN, this observation will be automatically removed from the result of the *inner join*.

If we go back to the `info` and `band_instruments` DataFrames, and use them as an example, you can see that only Paul and John are included on the result of an *inner join*. While in a *full join*, all musicians are included on the resulting DataFrame.

```
# An inner join between `info` and `band_instruments`:
info.join(band_instruments, on = 'name', how = 'inner')\
```



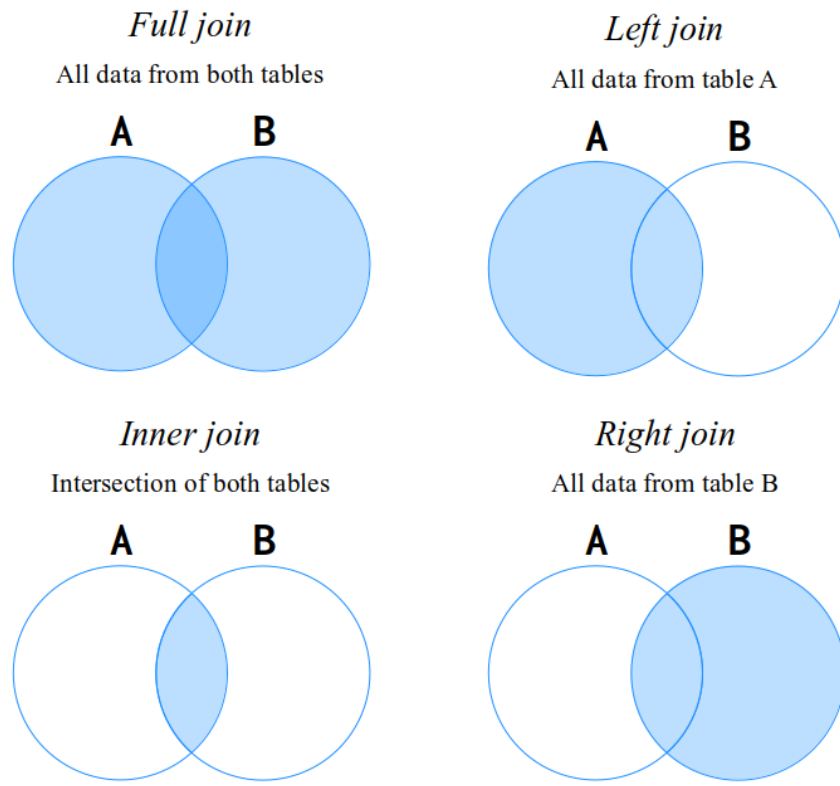


Figure 8.3: A visual representation for types of JOIN using numerical sets

```
.show()
```

```
+-----+-----+-----+-----+-----+
|name|   band|   born|children| plays|
+-----+-----+-----+-----+-----+
|John|Beatles|1940-09-10|   true|guitar|
|Paul|Beatles|1942-06-18|   true|  bass|
+-----+-----+-----+-----+-----+
```

```
# A full join between `info` and `band_instruments`:
info.join(band_instruments, on = 'name', how = 'full')\
.show()
```

```
+-----+-----+-----+-----+-----+
| name|   band|   born|children| plays|
+-----+-----+-----+-----+-----+
|George|   Beatles|1943-02-25|   true|  NULL|
| John|   Beatles|1940-09-10|   true|guitar|
| Keith|   NULL|   NULL|   NULL|guitar|
| Mick|Rolling Stones|1943-07-26|   true|  NULL|
| Paul|   Beatles|1942-06-18|   true|  bass|
| Ringo|   Beatles|1940-07-07|   true|  NULL|
+-----+-----+-----+-----+-----+
```

On the other hand, the *left join* and *right join* are kind of self-explanatory. On a *left join*, all the observations from the left DataFrame are kept intact on the resulting DataFrame of the JOIN, regardless of whether these observations were found or not on the right DataFrame. In contrast, an *right join* is the opposite of that. So, all observations from the right DataFrame are kept intact on the resulting DataFrame of the JOIN.

In pyspark, you can define the type of JOIN you want to use by setting the `how` argument at `join()` method. This argument accepts a string with the type of JOIN you want to use as input.

- `how = 'left'`: make a *left join*;
- `how = 'right'`: make a *right join*;
- `how = 'full'`: make a *full join*;
- `how = 'inner'`: make an *inner join*;
- `how = 'semi'`: make a *semi join*;
- `how = 'anti'`: make an *anti join*;

You can see on the list above, that pyspark do have two more types of JOINS, which are the *semi join* and *anti join*. These are “filtering types” of JOINS. Because they perform the matching process, and only filter the rows from table A (i.e. the DataFrame on the left side of the JOIN) based on the matches found on table B (i.e. the DataFrame on the right side of the JOIN).

In other words, these both types are used as a filter mechanism, and not as a merge mechanism. When you use these two types, instead of merging two DataFrames together, you are interested in filtering the rows of DataFrame A based on the existence of these rows in DataFrame B.

This is different from what we learned on *left*, *right*, *full* and *inner* types, because they do not only change which rows are included in the final result, but they also add the columns from table B into table A. Because of this behavior, these four main types are usually called as “additive types” of JOIN, since they are always adding data from table B into table A, i.e. they are merging the two tables together.

In more details, an *anti join* perform the exact opposite matching process of an *inner join*. This means that an *anti join* will always result in a new DataFrame that contains solely the observations that exists only on one DataFrame of the JOIN. In other words, the observations that are found on both tables are automatically removed from the resulting DataFrame of the JOIN. If we look at the example below, we can see that both John and Paul were removed from the resulting DataFrame of the *anti join*, because these two musicians are present on both DataFrames:

```
info.join(band_instruments, on = 'name', how = 'anti')\
.show()
```

name	band	born	children
Mick	Rolling Stones	1943-07-26	true
George	Beatles	1943-02-25	true
Ringo	Beatles	1940-07-07	true

In contrast, a *semi join* is equivalent to an *inner join*, with the difference that it does not adds the column from table B into table A. So this type of JOIN filter the rows from DataFrame A that also exists in DataFrame B. If an observation is found on both tables, this observation will appear on the resulting DataFrame.

```
info.join(band_instruments, on = 'name', how = 'semi')\
.show()
```

name	band	born	children
------	------	------	----------

```

+----+-----+-----+-----+
| John|Beatles|1940-09-10|    true|
| Paul|Beatles|1942-06-18|    true|
+----+-----+-----+-----+

```

Just to keep using our visual model of sets, on Figure 8.4 you can see the *semi* and *anti* JOIN types represented as numerical sets.

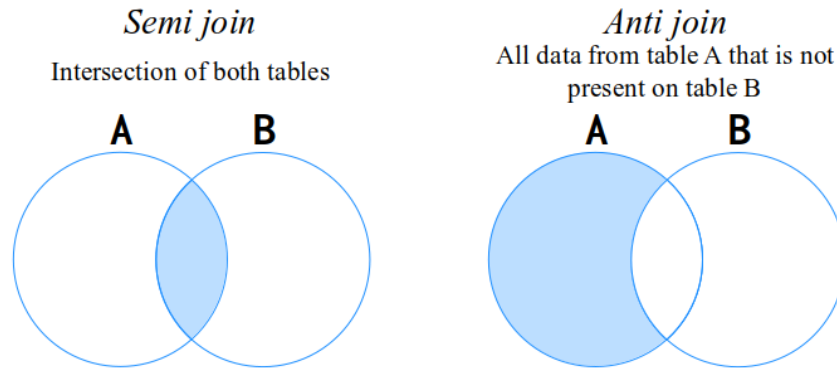


Figure 8.4: The two “filter types” of JOIN

### 8.4.3 A cross JOIN as the seventh type

We described six different types of JOINS on the previous section. But Spark also offers a seventh type of JOIN called *cross join*. This is a special type of JOIN that you can use by calling the `crossJoin()` DataFrame method.

In essence, a *cross join* returns, as output, the cartesian product between two DataFrames. It is similar to R functions `base::expand.grid()` or `dplyr::expand()`, and also, the Python equivalent `itertools.product()`.

This is a type of JOIN that you should avoid to use, specially if one (or both) of the DataFrames involved is a big DataFrame with thousands/millions of rows. Because a *cross join* will always produce a cartesian product between the two DataFrames involved. This means that, if DataFrame A contains  $x$  rows, and DataFrame B contains  $y$  rows, the end result of the *cross join* is a new DataFrame C that contains  $x \times y$  rows.

In other words, the number of rows in the output of a *cross join* can grow exponentially. For example, a *cross join* between a DataFrame of 1 thousand rows, and another DataFrame of 10 thousand of rows

(both are small DataFrames for the scale and sizes of a real-world big data environment), would produce a DataFrame with  $10^3 \times 10^4 = 10^7$ , that is, 10 million of rows as output.

In a big data environment, dealing with something that grows exponentially... it is never a good idea. So try to avoid a *cross join* and use him solely on very small DataFrames.

As an example, to apply a *cross join* between `info` and `band_instruments` DataFrames we can use the `crossJoin()` method, like in the example below:

```
info.crossJoin(band_instruments)\
.show()
```

```
+-----+-----+-----+-----+-----+-----+
| name|      band|    born|children| name| plays|
+-----+-----+-----+-----+-----+-----+
| Mick|Rolling Stones|1943-07-26|    true| John|guitar|
| Mick|Rolling Stones|1943-07-26|    true| Paul|  bass|
| Mick|Rolling Stones|1943-07-26|    true|Keith|guitar|
| John|      Beatles|1940-09-10|    true| John|guitar|
| John|      Beatles|1940-09-10|    true| Paul|  bass|
| John|      Beatles|1940-09-10|    true|Keith|guitar|
| Paul|      Beatles|1942-06-18|    true| John|guitar|
| Paul|      Beatles|1942-06-18|    true| Paul|  bass|
| Paul|      Beatles|1942-06-18|    true|Keith|guitar|
|George|      Beatles|1943-02-25|    true| John|guitar|
|George|      Beatles|1943-02-25|    true| Paul|  bass|
|George|      Beatles|1943-02-25|    true|Keith|guitar|
| Ringo|      Beatles|1940-07-07|    true| John|guitar|
| Ringo|      Beatles|1940-07-07|    true| Paul|  bass|
| Ringo|      Beatles|1940-07-07|    true|Keith|guitar|
+-----+-----+-----+-----+-----+-----+
```

A *cross join* is a special type of JOIN because it does not use “keys” and a matching process. It just computes every possible combination between the rows from both DataFrames. Because of the absence of these keys characteristics of a JOIN, many data analysts and engineers would not call a *cross join* as a type of JOIN (in other words, they would call it a type of something else). But regardless of our opinions, Spark decided to call this process as the *cross join*, so this is the way we are calling this process on this book.

## 8.5 Pivot operations

Pivot operations are extremely useful, and they are probably the main operation you can use to completely reformat your table. What these operations do is basically change the dimensions of your table. In other words, this kind of operation transform columns into rows, or vice versa.

As a comparison with other data frameworks, a pivot operation in Spark is the same operation performed by R functions `tidyr::pivot_longer()` and `tidyr::pivot_wider()` from the famous R framework `tidyverse`; or, the same as the `pivot()` and `melt()` methods from the Python framework `pandas`.

In Spark, pivot operations are performed by the `pivot()` DataFrame method, and by the `stack()` Spark SQL function. Pivot transformations are available in both directions. That is, you can transform either rows into columns (corresponds to `pivot()`), or, columns into rows (corresponds to `stack()`). Let's begin with `stack()`, and after that, we explain the `pivot()` method.

### 8.5.1 Transforming columns into rows

The `stack()` Spark SQL function allows you to transform columns into rows. In other words, you can make your DataFrame “longer” with this kind of operation, because you remove columns (“width”) from the table, and adds new rows (“height”). This gives an aspect of “longer” to your table, because after this operation, you table usually have more rows than columns.

As a first example, lets use the `religion` DataFrame, which you can re-create in your session with the source code below:

```
data = [
    ('Agnostic', 27, 34, 60),
    ('Atheist', 12, 27, 37),
    ('Buddhist', 27, 21, 30)
]
cols = ['religion', '<$10k', '$10k-$20k', '$20k-$30k']
religion = spark.createDataFrame(data, cols)
religion.show()
```

```
+-----+-----+-----+-----+
|religion|<$10k|$10k-$20k|$20k-$30k|
+-----+-----+-----+-----+
|Agnostic|  27|      34|      60|
|Atheist|  12|      27|      37|
|Buddhist| 27|      21|      30|
+-----+-----+-----+-----+
```

This DataFrame is showing us the average salary of people belonging to different religious groups. In each column of this DataFrame, you have data for a specific salary level (or range). This is a structure that can be easy and intuitive for some specific operations, but it also might impose some limitations, specially if you need to apply a vectorised operation over these salary ranges.

The basic unit of this DataFrame are the religious groups, and the salary ranges represents a characteristic of these groups. The different salary ranges are distributed across different columns. But what if we transformed these multiple columns into multiple rows? How can we accomplish that?

What we need to do, is to concentrate the labels (or the column names) of salary ranges into a single column, and move the respective values of the salary levels into another column. In other words, we need to create a column that contains the labels, and another column that contains the values. Figure 8.5 have a visual representation of this process:

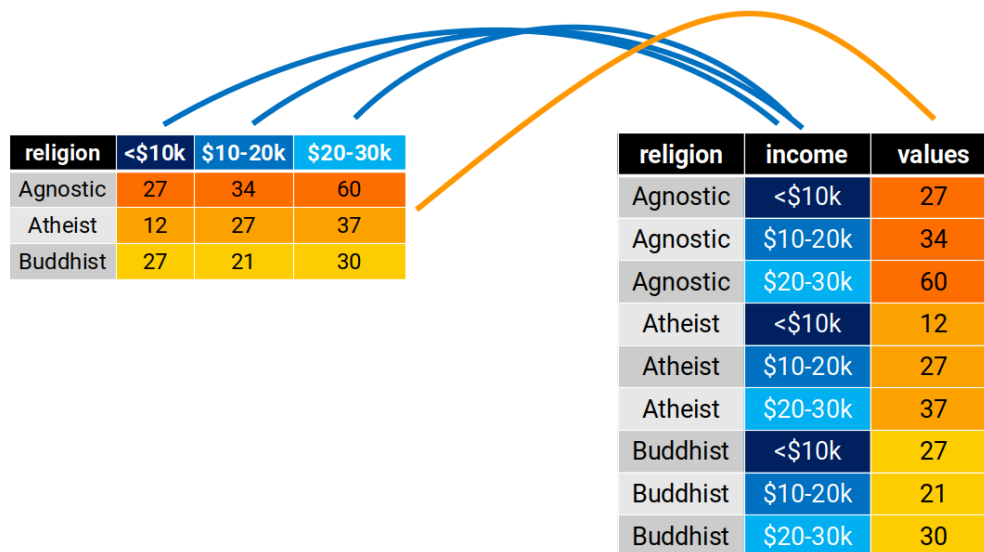


Figure 8.5: A visual representation of a pivot operation

Let's build this transformation in pyspark. First, remember that `stack()` is not a DataFrame method. It is a Spark SQL function. However, it is not an exported Spark SQL function, which means that you cannot import this function from the `pyspark.sql.function` module. This means that `stack()` will never be directly available in your python session to use.

So how do you use it? The answer is: use it inside Spark SQL! The `stack()` function is not available directly in python, but it is always available in Spark SQL, so all you need to do, is to use `stack()` inside functions and methods such as `expr()` (that I introduced at Section 7.6.2), or `sql()` to access Spark SQL functionality.

Now, the `stack()` function have two main arguments, which are the number of columns to transform into rows, and a sequence of key-value pairs that describes which columns will be transformed into rows, and the label values that corresponds to each column being transformed.

As a first example, the source code below replicates the transformation exposed at Figure 8.5:

```
from pyspark.sql.functions import expr
stack_expr = """
stack(3,
      '<$10k', `<$10k`,
      '$10k-$20k', ` $10k-$20k`,
      '$20k-$30k', ` $20k-$30k`
) AS (salary_range, avg_salary)
"""

longer_religion = religion\
    .select('religion', expr(stack_expr))

longer_religion.show()
```

religion	salary_range	avg_salary
Agnostic	<\$10k	27
Agnostic	\$10k-\$20k	34
Agnostic	\$20k-\$30k	60
Atheist	<\$10k	12
Atheist	\$10k-\$20k	27
Atheist	\$20k-\$30k	37
Buddhist	<\$10k	27
Buddhist	\$10k-\$20k	21
Buddhist	\$20k-\$30k	30

An important aspect about the `stack()` function, is that it always outputs two new columns (one column for the labels - or the keys, and another for the values). In the example above, these new columns are `salary_range` and `avg_salary`.

The first column identifies from which column (before the `stack()` operation) the value present at `avg_salary` came from. This means that this first column produced by `stack()` works as a column of labels or identifiers. These labels identify from which of the three transformed columns (<\$10k, \$10k-\$20k and \$20k-\$30k) the row value came from. In the visual representation exposed at Figure 8.5, this “labels column” is the income column.



On the other hand, the second column in the output of `stack()` contains the actual values that were present on the columns that were transformed. This “values column” in the example above corresponds to the column `avg_salary`, while in the visual representation exposed at Figure 8.5, it is the values column.

The first argument in `stack()` is always the number of columns that will be transformed by the function into rows. In our example, we have three columns that we want to transform, which are `<$10k`, `$10k-$20k` and `$20k-$30k`. That is why we have the number 3 as the first argument to `stack()`.

After that, we have a sequence of key-value pairs. In each pair, the value side (i.e. the right side) of the pair contains the name of the column that will be transformed, and the key side (i.e. the left side) of the pair contains the “label value”, or, in other words, which value represents, marks, label, or identifies the values that came from the column described in the right side of the pair.

Normally, you set the label value to be equivalent to the column name. That is, both sides of each pair are usually pretty much the same. But you can change this behaviour if you want. In the example below, all values that came from the `<$10k` are labeled as “Below \$10k”, while the values from the `$10k-$20k` column, are labeled in the output as “Between \$10k-\$20k”, etc.

```
stack_expr = """
stack(3,
      'Below $10k', `<$10k`,
      'Between $10k-$20k', `$10k-$20k`,
      'Between $20k-$30k', `$20k-$30k`
) AS (salary_range, avg_salary)
"""

religion\
  .select('religion', expr(stack_expr))\
  .show()
```

```
+-----+-----+-----+
|religion| salary_range|avg_salary|
+-----+-----+-----+
|Agnostic|    Below $10k|        27|
|Agnostic|Between $10k-$20k|        34|
|Agnostic|Between $20k-$30k|        60|
| Atheist|    Below $10k|        12|
| Atheist|Between $10k-$20k|        27|
| Atheist|Between $20k-$30k|        37|
|Buddhist|    Below $10k|        27|
|Buddhist|Between $10k-$20k|        21|
|Buddhist|Between $20k-$30k|        30|
+-----+-----+-----+
```

Furthermore, because the `stack()` function always outputs two new columns, if you want to rename these two new columns being created, to give them more readable and meaningful names, you always need to provide two new column names at once, inside a tuple, to the `AS` keyword.

In the example above, this tuple is `(salary_range, avg_salary)`. The first value in the tuple is the new name for the “labels column”, while the second value in the tuple, is the new name for the “values column”.

Now, differently from other Spark SQL functions, the `stack()` function should not be used inside the `withColumn()` method, and the reason for this is very simple: `stack()` always returns two new columns as output, but the `withColumn()` method can only create one column at a time.

This is why you get an `AnalysisException` error when you try to use `stack()` inside `withColumn()`, like in the example below:

```
religion\  
  .withColumn('salary_ranges', stack_expr)\  
  .show()
```

```
AnalysisException: The number of aliases supplied in the AS clause  
does not match the number of columns output by the UDTF expected  
2 aliases but got salary_ranges
```

## 8.5.2 Transforming rows into columns

On the other side, if you want to transform rows into columns in your Spark `DataFrame`, you can use the `pivot()` method. One key aspect of the `pivot()` method, is that it must always be used in conjunction with the `groupby()` method that we introduced at Section 5.10.4. In other words, `pivot()` does not work without `groupby()`.

You can think (or interpret) that the `groupby()` method does the job of defining (or identifying) which columns will be present in the output of `pivot()`. For example, if your `DataFrame` contains five columns, which are A, B, C, D and E; but you only listed columns A and C inside `groupby()`, this means that if you perform a pivot operation after that, the columns B, D and E will not be present in the output of `pivot()`. These three columns (B, D and E) will be automatically dropped during the pivot operation.

In contrast, the `pivot()` method does the job of identifying a single column containing the values that will be transformed into new columns. In other words, if you list the column `car_brands` inside `pivot()`, and, this column contains four unique values, for example, Audi, BMW, Jeep and Fiat, this means that, in the output of `pivot()`, four new columns will be created, named as Audi, BMW, Jeep and Fiat.

Therefore, we use `groupby()` to define the columns that will be kept intact on the output of the pivot operation; we use `pivot()` to mark the column that contains the rows that we want to transform into

new columns; at last, we must learn how to define which values will populate the new columns that will be created. Not only that, we also need to specify how these values will be calculated. And for that, we need to use an aggregating function.

This is really important, you can not do a pivot operation without aggregating the values that will compose (or populate) the new columns you are creating. Without it, Spark will not let you do the pivot operation using the `pivot()` method.

As a first example, let's return to the `religion` DataFrame. More specifically, to the `longer_religion` DataFrame, which is the pivoted version of the `religion` DataFrame that we created on the previous section, using the `stack()` function.

```
longer_religion.show()
```

```
+-----+-----+-----+
|religion|salary_range|avg_salary|
+-----+-----+-----+
|Agnostic|    <$10k|      27|
|Agnostic|   $10k-$20k|     34|
|Agnostic|   $20k-$30k|     60|
|Atheist|    <$10k|      12|
|Atheist|   $10k-$20k|     27|
|Atheist|   $20k-$30k|     37|
|Buddhist|    <$10k|      27|
|Buddhist|   $10k-$20k|     21|
|Buddhist|   $20k-$30k|     30|
+-----+-----+-----+
```

We can use this `longer_religion` DataFrame and the `pivot()` method to perform the inverse operation we described at Figure 8.5. In other words, we can re-create the `religion` DataFrame through `longer_religion` using the `pivot()` method. The source code below demonstrates how we could do such thing:

```
from pyspark.sql.functions import first

# Equivalent to the `religion` DataFrame:
longer_religion\
    .groupby('religion')\
    .pivot('salary_range')\
    .agg(first('avg_salary'))\
    .show()
```

```

+-----+-----+-----+-----+
|religion|$10k-$20k|$20k-$30k|<$10k|
+-----+-----+-----+-----+
|Agnostic|      34|      60|    27|
|Buddhist|      21|      30|    27|
|Atheist|      27|      37|    12|
+-----+-----+-----+-----+

```

In the example above, you can see the three core parts that we described: 1) use `groupby()` to define which columns will be preserved from the input DataFrame; 2) use `pivot()` to define which column will be used to transform rows into new columns; 3) the aggregating functions describing which values will be used, and how they are going to be calculated - `agg(first('avg_salary'))`. Figure 8.6 exposes these core parts in a visual manner.

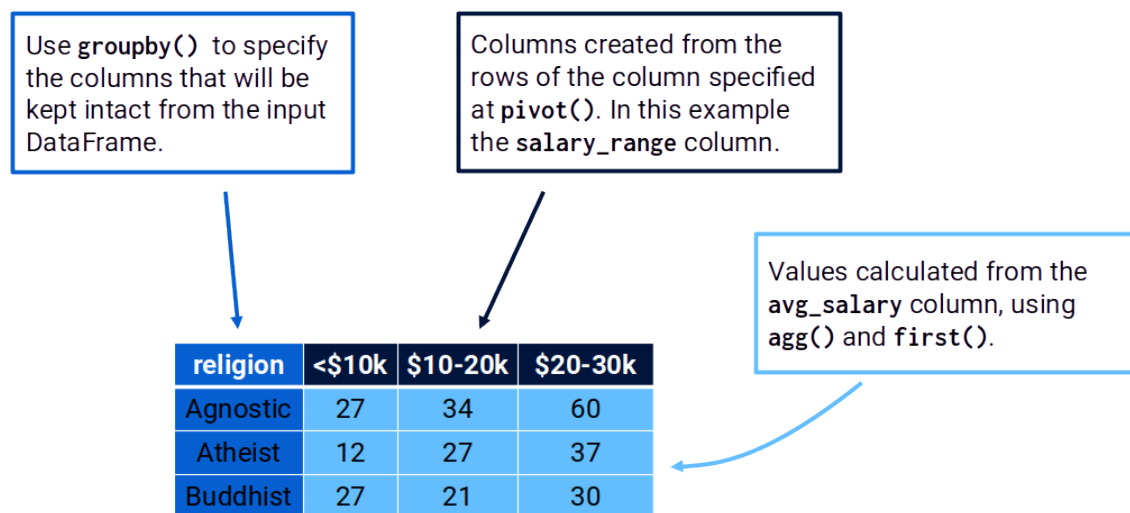


Figure 8.6: The three core parts that define the use of `pivot()`

When you use `pivot()`, you have to apply an aggregating function over the column from which you want to extract the values that will populate the new columns created from the pivot operation. This is a prerequisite, because Spark needs to know what he must do in case he finds two (or more) values that are mapped to the same cell in the pivot operation.

In the above example, we used the `first()` function to aggregate the values from the `avg_salary` column. With this function, we are telling Spark, that if it finds two (or more) values that are mapped

the same cell, then, Spark should pick the *first value if finds* in the input DataFrame, and simply ignore the remaining values.

Let's see an example. In the code chunk below, we are creating a new DataFrame called df:

```
data = [
    ('2023-05-01', 'Anne', 1, 15),
    ('2023-05-02', 'Mike', 2, 25),
    ('2023-05-02', 'Mike', 2, 34),
    ('2023-05-02', 'Mike', 2, 21),
    ('2023-05-03', 'Dani', 3, 18)
]
cols = ['date', 'name', 'id', 'value']
df = spark.createDataFrame(data, cols)
df.show()
```

```
+-----+-----+-----+
|      date|name| id|value|
+-----+-----+-----+
|2023-05-01|Anne|  1|   15|
|2023-05-02|Mike|  2|   25|
|2023-05-02|Mike|  2|   34|
|2023-05-02|Mike|  2|   21|
|2023-05-03|Dani|  3|   18|
+-----+-----+-----+
```

Let's suppose you want to transform the rows in the name column into new columns, and, populate these new columns with the values from the value column. Following what we discussed until now, we could do this by using the following source code:

```
df\
    .groupby('date', 'id')\
    .pivot('name')\
    .agg(first('value'))\
    .show()
```

```
+-----+-----+-----+
|      date| id|Anne|Dani|Mike|
+-----+-----+-----+
|2023-05-02|  2|NULL|NULL|  25|
|2023-05-03|  3|NULL|  18|NULL|
|2023-05-01|  1|  15|NULL|NULL|
```

```
+-----+-----+-----+-----+
```

However, there is a problem with this operation, because we lost some observations in the process. The specific combination ('2023-05-02', 'Mike') have three different values in the value column, which are 25, 34 and 21. But there is only a single cell in the new DataFrame (i.e. the output of the pivot operation) to hold these values. More specifically, the cell located at the first row in the fifth column.

In other words, Spark found three different values for a single cell (or single space), and this is always a problem. Spark cannot simply put three different values in a single cell. A Spark DataFrame just do not work that way. Every cell in a Spark DataFrame should always hold a single value, whatever that value is.

That is the exact problem that an aggregating function solves in a pivot operation. The aggregating function does the job of ensuring that a single value will be mapped to every new cell created from the pivot operation. Because an aggregating function is a function that aggregates (or that summarises) a set of values into a single value.

In the example above we used `first()` as our aggregating function. So when Spark encountered the three values from the combination ('2023-05-02', 'Mike'), it simply picked the first value from the three. That is why we find the value 25 at the first row in the fifth column.

We can change this behaviour by changing the aggregating function applied. In the example below, we are using `sum()`, and, as a result, we get now the value of 80 (which is the sum of values 25, 34 and 21) at the first row in the fifth column.

```
from pyspark.sql.functions import sum
df\
    .groupby('date', 'id')\
    .pivot('name')\
    .agg(sum('value'))\
    .show()
```

```
+-----+-----+-----+-----+
|      date| id|Anne|Dani|Mike|
+-----+-----+-----+-----+
|2023-05-02|  2|NULL|NULL|  80|
|2023-05-03|  3|NULL|  18|NULL|
|2023-05-01|  1|  15|NULL|NULL|
+-----+-----+-----+-----+
```

Now, depending on the data and the structure from your DataFrame, Spark might never encounter a situation where it finds two (or more) values mapped to the same cell in a pivot operation. On this specific case, the output from the pivot operation will likely be the same for many different aggregating

functions. In other words, the choice of the aggregating function you want to use might be irrelevant over this specific situation.

For example, in the `longer_religion` example that we showed before, I could use the `last()` aggregating function (instead of `first()`), and get the exact same result as before. Because in this specific situation, Spark does not find any case of two (or more) values mapped to the same cell in the output DataFrame.

```
from pyspark.sql.functions import last
longer_religion\
    .groupby('religion')\
    .pivot('salary_range')\
    .agg(last('avg_salary'))\
    .show()
```

```
+-----+-----+-----+-----+
|religion|$10k-$20k|$20k-$30k|<$10k|
+-----+-----+-----+-----+
|Agnostic|      34|      60|    27|
|Buddhist|      21|      30|    27|
|Atheist|      27|      37|    12|
+-----+-----+-----+-----+
```

## 8.6 Collecting and explode operations

You can retract or extend vertically your DataFrame, by nesting multiple rows into a single row, or, the inverse, which is unnesting (or exploding) one single row into multiple rows. The R tidyverse framework is probably the only data framework that have a defined name for this kind of operation, which are called “nesting” and “unnesting”.

In Spark, you perform this kind of operations by using the `collect_list()`, `collect_set()` and `explode()` functions, which all comes from the `pyspark.sql.functions` module. The `collect_list()` and `collect_set()` functions are used for retracting (or nesting) your DataFrame, while the `explode()` function is used for extending (or unnesting).

As a quick comparison, `explode()` is very similar to the R function `tidyr::unnest_longer()` from the tidyverse framework, and also, similar to the Python method `explode()` in the pandas framework. In the other hand, `collect_list()` and `collect_set()` functions does a similar job to the R function `tidyr::nest()`.

### 8.6.1 Expanding (or unnesting) with explode()

As an example, we have the employees DataFrame below. Each row in this DataFrame describes an employee. The knowledge column describes which programming language the employee have experience with, and, the employee\_attrs column contains dictionaries with general attributes of each employee (such it's department and it's name).

```
data = [
    (1, ["R", "Python"], {'dep': 'PR', 'name': 'Anne'}),
    (2, ["Scala"], {'dep': 'PM', 'name': 'Mike'}),
    (3, ["Java", "Python"], {'dep': 'HF', 'name': 'Sam'})
]
columns = ["employee_id", "knowledge", "employee_attrs"]
employees = spark.createDataFrame(data, columns)
employees.show()
```

```
+-----+-----+-----+
|employee_id|  knowledge| employee_attrs|
+-----+-----+-----+
|          1| [R, Python]|{name -> Anne, de...|
|          2|    [Scala]|{name -> Mike, de...|
|          3|[Java, Python]|{name -> Sam, dep...|
+-----+-----+-----+
```

You can use the printSchema() method that we introduced at Section 3.9 to see the schema of the DataFrame. You can see in the result below that knowledge is a column of arrays of strings (i.e. ArrayType), while employee\_attrs is a column of maps (i.e. MapType).

```
employees.printSchema()
```

```
root
 |-- employee_id: long (nullable = true)
 |-- knowledge: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- employee_attrs: map (nullable = true)
 |    |-- key: string
 |    |-- value: string (valueContainsNull = true)
```

Suppose you wanted to calculate the number of employees that have experience in each programming language. To do that, it would be great to transform the knowledge column into a column of strings. Why? Because it would make the counting of the employees easier.



In other words, if an employee knows, for example, 3 different programming languages, it would be better to have 3 different rows that references this same employee, instead of having a single row that contains an array of three elements, like in the employees DataFrame. We can transform the current DataFrame into this new format by using the `explode()` function.

When you apply the `explode()` function over a column of arrays, this function will create a new row for each element in each array it finds in the column. For example, the employee of ID 1 knows two programming languages (R and Python). As a result, when we apply `explode()` over the knowledge column, two rows are created for the employee of ID 1. As you can see in the example below:

```
from pyspark.sql.functions import explode
explode_array = employees\
    .select(
        'employee_id',
        explode('knowledge')
    )

explode_array.show()
```

```
+-----+-----+
|employee_id|  col|
+-----+-----+
|          1|    R|
|          1|Python|
|          2|  Scala|
|          3|   Java|
|          3|Python|
+-----+-----+
```

On the other hand, instead of arrays, the behaviour of `explode()` is slightly different when you apply it over a column of maps, such as `employee_attrs`. Because each element in a map have two components: a key and a value. As a consequence, when you apply `explode()` over a column of maps, each element in the map generates two different rows, which are stored in two separated columns, called key and value.

Take the result below as an example. First, we have two new columns that were not present before (key and value). Each row in the key column represents the key for an element in the input map. While the value column represents the values of those elements.

```
explode_map = employees\
    .select(
        'employee_id',
```

```

        explode('employee_attrs')
    )

explode_map.show()

```

```

+-----+-----+
|employee_id| key|value|
+-----+-----+
|          1| name| Anne|
|          1| dep|  PR|
|          2| name| Mike|
|          2| dep|  PM|
|          3| name|  Sam|
|          3| dep|  HF|
+-----+-----+

```

This kind of output is powerful, specially with pivot operations, because you can easily organize all the data found in a column of maps into a series of new columns, like this:

```

explode_map\
    .groupby('employee_id')\
    .pivot('key')\
    .agg(first('value'))\
    .show()

```

```

+-----+-----+
|employee_id| dep| name|
+-----+-----+
|          1| PR| Anne|
|          2| PM| Mike|
|          3| HF|  Sam|
+-----+-----+

```

### 8.6.2 The different versions of `explode()`

The `explode()` function have three brother functions, which are `explode_outer()`, `posexplode()` and `posexplode_outer()`. All of these functions have very small differences between them. However, these differences might be important/useful to you.

First, the difference between `explode()` and `explode_outer()` is on the null values. If an array contains a null value, `explode()` will ignore this null value. In other words, `explode()` does not create a

new row for any null value. In contrast, `explode_outer()` does create a new row even if the value is null.

For example, if `explode()` encounter an array with 5 elements where 2 of them are null values, then, `explode()` will output 3 new rows. The 2 null values in the input array are automatically ignored. However, if `explode_outer()` encountered the same array, then, it would output 5 new rows, no matter which values are inside this input array.

Second, the difference between `explode()` and `posexplode()`, is that `posexplode()` also returns the index that identifies the position in the input array where each value is. In other words, if we applied the `posexplode()` over the knowledge column of employee DataFrame, we would get a new column called `pos`. By looking at this column `pos`, we could see that the value "Python" for the employee of ID 1, was on the position of index 1 on the original array that the function encountered in the original knowledge column.

```
from pyspark.sql.functions import posexplode
employees\
    .select(
        'employee_id',
        posexplode('knowledge')
    )\
    .show()
```

```
+-----+---+-----+
|employee_id|pos|  col|
+-----+---+-----+
|          1|  0|    R|
|          1|  1|Python|
|          2|  0|  Scala|
|          3|  0|   Java|
|          3|  1|Python|
+-----+---+-----+
```

Third, as you can probably imagine, the function `posexplode_outer()` incorporates the two visions from the previous brothers. So, not only `posexplode_outer()` creates a row for each element in the array, no matter if this element is a null value or not, but it also returns the index that identifies the position in the input array where each value is.

### 8.6.3 Retracting (or nesting) with `collect_list()` and `collect_set()`

In resume, you use the `collect_list()` and `collect_set()` functions to retract your DataFrame. That is, to reduce the number of rows of the DataFrame, while keeping the same amount of information.

To do this you just aggregate your DataFrame, using the `agg()` method, and, apply the `collect_list()` or `collect_set()` function over the columns you want to retract (or nest). You likely want to use the `groupby()` method as well in this case, to perform an aggregation per group.

Because if you do not define any group in this situation, you will aggregate the entire DataFrame into a single value. This means, that you would get as output, a new DataFrame with only a single row, and, all rows from the input DataFrame would be condensed inside this single row in the output DataFrame.

In essence, what the `collect_list()` function do is collect a set of rows and store it in an array. The `collect_set()` function does the same thing, with the difference that it stores this set of rows in a set (which is an array of unique values). In other words, `collect_set()` collects a set of rows, then, it removes all duplicated rows in this set, then, it stores the remaining values in an array.

To demonstrate the `collect_list()` and `collect_set()` functions, we can use the `supermarket_sales` DataFrame that we introduced at Section 8.1 as an example:

```
supermarket_sales.show()
```

```
+-----+-----+-----+-----+
|transaction_id|product_name|quantity|price|
+-----+-----+-----+-----+
|      T001|      Apple|      5|  1.2|
|      T001|      Apple|      5|  1.2|
|      T002|     Banana|      3|  0.8|
|      T004|      Mango|      2|  2.0|
|      T004|      Mango|      2|  2.0|
|      T004|      Mango|      2|  2.0|
|      T005|     Grapes|      1|  3.5|
|      T006|      Apple|      2|  1.2|
|      T006|      Apple|      1|  1.2|
|      T007|     Banana|      4|  0.8|
|      T008|      Apple|      3|  1.2|
+-----+-----+-----+-----+
```

Suppose you wanted a new DataFrame that had a single row for each `transaction_id` without losing any amount of information from the above DataFrame. We could do this by using the `collect_list()` function, like in the example below:

```
from pyspark.sql.functions import collect_list
supermarket_sales\
    .groupby('transaction_id')\
    .agg(
        collect_list('product_name').alias('product_name'),
```

```

        collect_list('quantity').alias('quantity'),
        collect_list('price').alias('price')
    )\
    .show()

```

transaction_id	product_name	quantity	price
T001	[Apple, Apple]	[5, 5]	[1.2, 1.2]
T002	[Banana]	[3]	[0.8]
T004	[Mango, Mango, Ma...]	[2, 2, 2]	[2.0, 2.0, 2.0]
T005	[Grapes]	[1]	[3.5]
T006	[Apple, Apple]	[2, 1]	[1.2, 1.2]
T007	[Banana]	[4]	[0.8]
T008	[Apple]	[3]	[1.2]

The expression `groupby('transaction_id')` ensures that we have (on the output DataFrame) a single row for each unique value in the `transaction_id` column. While the `collect_list()` function does the job of condensing all the information of the remaining columns into a single row for each `transaction_id`.

Now, if we use `collect_set()` instead of `collect_list()`, we would get a slightly different result. Because, as we described before, the `collect_set()` function removes all duplicated values found in the set of rows it collects.

```

from pyspark.sql.functions import collect_set
supermarket_sales\
    .groupby('transaction_id')\
    .agg(
        collect_set('product_name').alias('product_name'),
        collect_set('quantity').alias('quantity'),
        collect_set('price').alias('price')
    )\
    .show()

```

transaction_id	product_name	quantity	price
T001	[Apple]	[5]	[1.2]
T002	[Banana]	[3]	[0.8]

	T004	[Mango]	[2]	[2.0]
	T005	[Grapes]	[1]	[3.5]
	T006	[Apple]	[1, 2]	[1.2]
	T007	[Banana]	[4]	[0.8]
	T008	[Apple]	[3]	[1.2]
+-----+-----+-----+-----+				

## 9 Exporting data out of Spark

After you transform your `DataFrame` and generate the results you want, you might need to actually export these results out of Spark, so you can:

- send the exported data to an external API.
- send these results to your manager or client.
- send the exported data to an ingest process that feeds some database.

### 9.1 The write object as the main entrypoint

Every Spark session you start has an built-in read object that you can use to read data and import it into Spark (this object was described at Section 6.1), and the same applies to writing data out of Spark. That is, Spark also offers a write object that you can use to write/output data out of Spark.

But in contrast to the read object, which is available through the `SparkSession` object (`spark`), this write object is available through the `write` method of any `DataFrame` object. In other words, every `DataFrame` you create in Spark has a built-in write object that you can use to write/export the data present in this `DataFrame` out of Spark.

As an example, let's use the `transf` `DataFrame` that I presented at Chapter 5. The `write` method of the `transf` `DataFrame` object is the main entrypoint to all the facilities that Spark offers to write/export `transf`'s data to somewhere else.

```
transf.write
```

```
<pyspark.sql.readwriter.DataFrameWriter at 0x7f662d142fe0>
```

This write object is very similar in structure to the read object. Essentially, this write object has a collection of *write engines*. Each write engine is specialized in writing data into a specific file format. So you have an engine for CSV files, another engine for JSON files, another for Parquet files, etc.

Every write object has the following methods:

- `mode()`: set the mode of the write process. This affects how the data will be written to the files, and how the process will behave if exceptions (or errors) are raised during runtime.

- `option()`: set an option to be used in the write process. This option might be specific to the write engine used, or, might be an option that is global to the write process (i.e. an option that does not depend of the chosen engine).
- `csv()`: the write engine to export data to CSV files.
- `json()`: the write engine to export data to JSON files.
- `parquet()`: the write engine to export data to Parquet files.
- `orc()`: the write engine to export data to ORC files.
- `text()`: the write engine to export data to text files.
- `jdbc()`: saves the data of the current DataFrame into a database using the JDBC API.

## 9.2 Exporting the transf DataFrame

As a first example on how to export data out of Spark, I will export the data from the `transf` DataFrame. Over the next sections, I will cover individual aspects that influences this write/export process. You should know and consider each of these individual aspects when exporting your data.

### 9.2.1 Quick export to a CSV file

Lets begin with a quick example of exporting the Spark data to a CSV file. For this job, we need to use the write engine for CSV files, which is the `csv()` method from the write object.

The **first (and main) argument to all write engines** available in Spark is a path to a folder where you want to store the exported files. This means that (whatever write engine you use) Spark will always write the files (with the exported data) inside a folder.

Spark needs to use a folder to write the data. Because it generates some extra files during the process that serves as “placeholders” or as “statuses”. That is why Spark needs to create a folder, to store all of these different files together during the process.

In the example below, I decided to write this data into a folder called `transf_export`.

```
transf.write.csv("transf_export")
```

Now, after I executed the above command, if I take a look at my current working directory, I will see the `transf_export` folder that was created by Spark.

```
from pathlib import Path
current_directory = Path(".")
folders_in_current_directory = [
    str(item)
    for item in current_directory.iterdir()]
```



```

        if item.is_dir()
    ]

    print(folders_in_current_directory)

```

```
['metastore_db', 'transf_export']
```

And if I look inside this `transf_export` folder I will see two files. One is the placeholder file (`_SUCCESS`), and the other, is a CSV file containing the exported data (`part-*.csv`).

```

export_folder = Path("transf_export")
files = [str(x.name) for x in export_folder.iterdir()]
print(files)

```

```
['part-00000-a4ee2ff4-4b7f-499e-a904-cec8d524ac56-c000.csv', '_SUCCESS']
```

We can see this file structure by using the [tree command line utility](#)<sup>1</sup> to build a diagram of this file structure:

```
Terminal$ tree transf_export
```

```

transf_export
├── part-00000-a4ee2ff4-4b7f-499e-a904-cec8d524ac56-c000.csv
└── _SUCCESS

```

## 9.2.2 Setting the write mode

You can set the mode of a write process by using the `mode()` method. This “mode of the write process” affects specially the behavior of the process when files for this particular DataFrame you trying to export already exists in your file system.

There are four write modes available in Spark:

- `append`: will append the exported data to existing files of this specific DataFrame.
- `overwrite`: will overwrite the data inside existing files of this specific DataFrame with the data that is being currently exported.
- `error or errorifexists`: will throw an exception in case already existing files for this specific DataFrame are found.

---

<sup>1</sup><https://www.geeksforgeeks.org/tree-command-unixlinux/>

- ignore: silently ignore/abort this write operation in case already existing files for this specific DataFrame are found.

If we set the write mode to overwrite, this means that every time we execute the command below, the files inside the folder `transf_export` are rewritten from scratch. Everytime we export the data, the files `part-*` inside the folder are rewritten to contain the most fresh data from `transf` DataFrame.

```
transf.write\
    .mode("overwrite")\
    .csv("transf_export")
```

However, if we set the write mode to error, and run the command again, then an error will be raised to indicate that the folder (`transf_export`) where we are trying to write the files already exists.

```
transf.write\
    .mode("error")\
    .csv("transf_export")
```

```
AnalysisException: [PATH_ALREADY_EXISTS]
Path file:/home/pedro/Documentos/Projetos/Livros/Introd-pyspark/Chapters/transf_
export
already exists. Set mode as "overwrite" to overwrite the existing path.
```

In contrast, if we set the write mode to append, then the current data of `transf` is appended (or “added”) to the folder `transf_export`.

```
transf.write\
    .mode("append")\
    .csv("transf_export")
```

Now, if I take a look at the contents of the `transf_export` folder, I will see now two `part-*` files instead of just one. Both files have the same size (around 218 kb) because they both contain the same data, or the same lines from the `transf` DataFrame.

```
Terminal$ tree transf_export
```

```
transf_export
├── part-00000-a4ee2ff4-4b7f-499e-a904-cec8d524ac56-c000.csv
├── part-00000-ffcc7487-fc60-403b-a815-a1dd56894062-c000.csv
└── _SUCCESS
```

This means that the data is currently duplicated inside the `transf_export` folder. We can see this duplication by looking at the number of rows of the DataFrame contained inside `transf_export`. We can use `spark.read.load()` to quickly load the contents of the `transf_export` folder into a new DataFrame, and use `count()` method to see the number of rows.

```
df = spark.read.load(
    "transf_export",
    format = "csv",
    header = False
)
df.count()
```

4842

The result above show us that the folder `transf_export` currently contains 4842 rows of data. This is the exact double of number of rows in the `transf` DataFrame, which have 2421 rows.

```
transf.count()
```

2421

So, in resume, the difference between write mode `overwrite` and `append`, is that `overwrite` causes Spark to erase the contents of `transf_export`, before it starts to write the current data into the folder. This way, Spark exports the most recent version of the data stored inside the DataFrame. In contrast, `append` simply appends (or adds) new files to the folder `transf_export` with the most recent version of the data stored inside the DataFrame.

At Section [7.2.1.6](#) (or more specifically, at Figure [7.1](#)) we presented this difference visually. So, in case you don't understood fully the difference between these two write modes, you can comeback at Section [7.2.1.6](#) and check Figure [7.1](#) to see if it clears your understanding. OBS: `save modes` = write modes.

### 9.2.3 Setting write options

Each person might have different needs, and also, each file format (or each write engine) have its particularities or advantages that you may need to exploit. As a consequence, you might need to set some options to customize the writing process to fit into your needs.

You can set options for the write process using the `option()` method of the write object. This method works with key value pairs. Inside this method, you provide the a key that identifies the option you want to set, and the value you want to give to this option.

For CSV files, an option that is very popular is the `sep` option, that corresponds to the separator character of the CSV. This is a special character inside the CSV file that separates each column field.

As an example, if we wanted to build a CSV file which uses the semicolon (; - which is the european standard for CSV files) as the separator character, instead of the comma (, - which is the american standard for CSV files), we just need to set the `sep` option to ;, like this:

```
transf\  
  .write\  
  .mode("overwrite")\  
  .option("sep", ";")\  
  .csv("transf_export")
```

Each file format (or each write engine) have different options that are specific (or characteristic) to the file format itself. For example, JSON and CSV files are text file formats, and because of that, one key aspect to them is the encoding of the text that is being stored inside these files. So both write engines for these file formats (`csv()` and `json()`) have an option called `encoding` that you can use to change the encoding being used to write the data into these files.

In the example below, we are asking Spark to write a CSV file using the Latin1 encoding (ISO-8859-1).

```
transf\  
  .write\  
  .mode("overwrite")\  
  .option("encoding", "ISO-8859-1")\  
  .csv("transf_export")
```

Is worth mentioning that the `option()` method sets one option at a time. So if you need to set various write options, you just stack `option()` calls on top of each other. In each call, you set a different option. Like in the example below where we are setting options `sep`, `encoding` and `header`:

```
transf\  
  .write\  
  .mode("overwrite")\  
  .option("sep", ";")\  
  .option("encoding", "UTF-8")\  
  .option("header", True)\  
  .csv("transf_export")
```

If you want to see the full list of options for each write engine, the documentation of Spark have a table with the complete list of options available at each write engine<sup>2</sup>.

---

<sup>2</sup><https://spark.apache.org/docs/latest/sql-data-sources-csv.html#data-source-option>.

## 9.3 Number of partitions determines the number of files generated

As I explained at Section 3.2, every DataFrame that exists in Spark is a **distributed** DataFrame, meaning that this DataFrame is divided into multiple pieces (that we call *partitions*), and these pieces are spread across the nodes in the Spark cluster.

In other words, each machine that is present in the Spark cluster, contains some partitions (or some pieces) of the total DataFrame. But why we are discussing partitions here? Is because the number of partitions of your DataFrame determines the number of files written by Spark when you export the data using the write method.

On the previous examples across Section 9.2, when we exported the `transf` DataFrame into CSV files, only one single CSV file was generated inside the `transf_exported` folder. That is because the `transf` DataFrame have only one single partition, as the code below demonstrates:

```
transf.rdd.getNumPartitions()
```

```
1
```

That means that all the data from `transf` DataFrame is concentrated into a single partition. Having that in mind, we could say that Spark decided in this specific case to not actually distribute the data of `transf`. Because all of its data is concentrated into one single place.

But what would happen if the `transf` DataFrame was splitted across 5 different partitions? What would happen then? In that case, if the `transf` DataFrame had 5 different partitions, and I ran the command `transf.write.csv("transf_export")` to export its data into CSV files, then, 5 different CSV files would be written by Spark inside the folder `transf_export`. One CSV file for each existing partition of the DataFrame.

The same goes for any other file format, or any write engine that you might use in Spark. Each file generated by the write process contains the data from a specific partition of the DataFrame.

### 9.3.1 Avoid exporting too much data into a single file

Spark will always try to organize your DataFrame into a *partition distribution* that yields the best performance in any data processing. Usually in production environments, we have huge amounts of data, and a single partition distribution is rarely the case that yields the best performance in these environments.

That is why most existing Spark DataFrames in production environments are splitted into multiple partitions across the Spark cluster. This means that Spark DataFrames that are by default concentrated into one single partition (like the `transf` DataFrame in the examples of this book) are very, very rare to find in the production environments.

As a consequence, if you really need to export your data into a single static file in a production environment, you will likely need to:

1. repartition your Spark DataFrame. That is, to reorganize the partitions of this DataFrame, so that all of its data get concentrated into a single partition.
2. or you continue with the write process anyway, and then later, after the write process is finished, you merge all of the generated files together with some other tool, like pandas, or polars, or the tidyverse.

The option 2 above is a little out of the scope of this book, so I will not explain it further here. But if you really need to export all the data from your Spark DataFrame into a single static file (whatever is the file format you choose), and you choose to follow option 1, then, you need to perform a repartition operation to concentrate all data from your Spark DataFrame into a single partition.

Is worth mentioning that **I strongly advise against this option 1**. Because option 1 may cause some serious bottlenecks in your data pipeline, depending specially on the size of the DataFrame you are trying to export.

In more details, when you do not perform any repartition operation, that is, when you just write your DataFrame as is, without touching in the existing partitions, then, the write process is a narrow transformation, as I explained at Section 5.3. Because each partition is exported into a single and separate file that is independent from the others.

This is really important, because narrow transformations are much more predictable and are more easily scaled than wide transformations. As a result, Spark tends to scale and perform better when dealing with narrow transformations.

However, when you do perform a repartition operation to concentrate all the data into a single partition, then, three things happen:

1. the write process becomes a wide transformation, because all partitions needs to be merged together, and as a consequence, all nodes in the cluster needs to send their data to a single place (which is usually the driver node of the cluster).
2. a high amount of partition shuffles can happen inside the cluster, and if they do happen, then, depending on the amount of data that needs to be “shuffled” accross the cluster, this may cause some serious slowdown in the processing.
3. depending on the size of all partitions merged together, the risks for an “out of memory” error to be raised during the process scales rapidly.

So you should be aware of these risks above, and always try to avoid using the option 1. Actually, you should avoid as much as possible the need to write all the data into a single static file! Is best for you to just write the data using the default number of partitions that Spark choose for your DataFrame.

But anyway, if you really cannot avoid this need, and if you have, for example, a sales DataFrame you want to export, and this DataFrame contains 4 partitions:

```
sales.rdd.getNumPartitions()
```

4

And you want to perform a repartition operation over this DataFrame to export its data into a single static file, you can do so by using the `coalesce()` DataFrame method. Just provide the number 1 to this method, and all of the partitions will be reorganized into a single partition:

```
sales\  
  .coalesce(1)\  
  .rdd\  
  .getNumPartitions()
```

1

Having that in mind, the entire source code to export the DataFrame into a single static file would be something like this:

```
sales\  
  .coalesce(1)\  
  .write\  
  .mode("overwrite")\  
  .csv("sales_export")
```

## 9.4 Transforming to a Pandas DataFrame as a way to export data

In case you don't know about this, Spark offers an API that you can use to quickly convert your Spark DataFrames into a pandas DataFrame. This might be extremely useful for a number of reasons:

- your colleague might be much more familiar with pandas, and work more productively with it than pyspark.
- you might need to feed this data into an existing data pipeline that uses pandas extensively.
- with pandas you can easily export this data into Excel files (.xlsx)<sup>3</sup>, which are not easily available in Spark.

---

<sup>3</sup>Actually, there is a Spark plugin available that is capable of exporting data from Spark directly into Excel files. But you need to install this plugin separately, since it does not come with Spark from the factory: <https://github.com/crealytics/spark-excel>.

To convert an existing Spark DataFrame into a pandas DataFrame, all you need to do is to call the `toPandas()` method of your Spark DataFrame, and you will get a pandas DataFrame as output, like in the example below:

```
as_pandas_df = transf.toPandas()
type(as_pandas_df)
```

```
pandas.core.frame.DataFrame
```

But you should be careful with this method, because when you transform your Spark DataFrame into a pandas DataFrame you eliminate the distributed aspect of it. As a result, all the data from your DataFrame needs to be loaded into a single place (which is usually the driver's memory).

Because of that, using this `toPandas()` method might cause very similar issues as the ones discussed at Section 9.3. In other words, you might face the same slowdowns caused by doing a repartition to concentrate all the data into a single partition.

So, as the Spark documentation itself suggests, you should use this `toPandas()` method only if you know that your DataFrame is small enough to fit into the driver's memory.

## 9.5 The `collect()` method as a way to export data

The `collect()` DataFrame method exports the DataFrame's data from Spark into a Python native object, more specifically, into a normal Python list. To some extent, this is a viable way to export data from Spark.

Because by making this data from Spark available as a normal/standard Python object, many new possibilities become open for us. Such as:

- sending this data to another location via HTTP requests using the `request` Python package.
- sending this data by email using the `email` built-in Python package.
- sending this data by SFTP protocol with the `paramiko` Python package.
- sending this data to a cloud storage, such as Amazon S3 (using the `boto3` Python package).

By having the DataFrame's data easily available to Python as a Python list, we can do virtually anything with this data. We can use this data in basically anything that Python is capable of doing.

Just as a simple example, I needed to send the `transf` data to an fictitious endpoint using a POST HTTP request, the source code would probably be something similar to this:

```
import requests

dataframe_rows = transf.collect()
```



```
url = 'https://example.com/api/v1/transf'
for row in dataframe_rows:
    row_as_dict = row.asDict()
    requests.post(url, data = row_as_dict)
```

# 10 Tools for string manipulation

Many of the world's data is represented (or stored) as text (or string variables). As a consequence, is very important to know the tools available to process and transform this kind of data, in any platform you use. In this chapter, we will focus on these tools.

Most of the functionality available in pyspark to process text data comes from functions available at the `pyspark.sql.functions` module. This means that processing and transforming text data in Spark usually involves applying a function on a column of a Spark DataFrame (by using DataFrame methods such as `withColumn()` and `select()`).

## 10.1 The logs DataFrame

Over the next examples in this chapter, we will use the logs DataFrame, which contains various log messages registered at a fictitious IP address. The data that represents this DataFrame is freely available through the `logs.json` file, which you can download from the official repository of this book<sup>1</sup>.

Each line of this JSON file contains a message that was recorded by the logger of a fictitious system. Each log message have three main parts, which are: 1) the type of message (warning - WARN, information - INFO, error - ERROR); 2) timestamp of the event; 3) the content of the message. In the example below, we have an example of message:

[INFO]: 2022-09-05 03:35:01.43 Looking for workers at South America region;

To import `logs.json` file into a Spark DataFrame, I can use the following code:

```
path = '../Data/logs.json'
logs = spark.read.json(path)
n_truncate = 50
logs.show(5, truncate = n_truncate)
```

```
+-----+-----+
|          ip|          message|
+-----+-----+
| 1.0.104.27 | [INFO]: 2022-09-05 03:35:01.43 Looking for work...|
```

<sup>1</sup><https://github.com/pedropark99/Introd-pyspark/tree/main/Data>

```
| 1.0.104.27 |[WARN]: 2022-09-05 03:35:58.007 Workers are una...|
| 1.0.104.27 |[INFO]: 2022-09-05 03:40:59.054 Looking for wor...|
| 1.0.104.27 |[INFO]: 2022-09-05 03:42:24 3 Workers were acqu...|
| 1.0.104.27 |[INFO]: 2022-09-05 03:42:37 Initializing instan...|
+-----+
only showing top 5 rows
```

By default, when we use the `show()` action to see the contents of our Spark DataFrame, Spark will always truncate (or cut) any value in the DataFrame that is more than 20 characters long. Since the logs messages in the `logs.json` file are usually much longer than 20 characters, I am using the `truncate` argument of `show()` in the example above, to avoid this behaviour.

By setting this argument to 50, I am asking Spark to truncate (or cut) values at the 50th character (instead of the 20th). By doing this, you (reader) can actually see a much more significant part of the logs messages in the result above.

## 10.2 Changing the case of letters in a string

Probably the most basic string transformation that exists is to change the case of the letters (or characters) that compose the string. That is, to raise specific letters to upper-case, or reduce them to lower-case, and vice-versa.

As a first example, lets go back to the `logs` DataFrame, and try to change all messages in this DataFrame to lower case, upper case and title case, by using the `lower()`, `upper()`, and `initcap()` functions from the `pyspark.sql.functions` module.

```
from pyspark.sql.functions import (
    lower,
    upper,
    initcap
)

m = logs.select('message')
# Change to lower case:
m.withColumn('message', lower('message'))\
  .show(5, truncate = n_truncate)
```

```
+-----+
|                                     message|
+-----+
|[info]: 2022-09-05 03:35:01.43 looking for work...|
```

```
|[warn]: 2022-09-05 03:35:58.007 workers are una...|
|[info]: 2022-09-05 03:40:59.054 looking for wor...|
|[info]: 2022-09-05 03:42:24 3 workers were acqu...|
|[info]: 2022-09-05 03:42:37 initializing instan...|
+-----+
only showing top 5 rows
```

```
# Change to upper case:
m.withColumn('message', upper('message'))\
  .show(5, truncate = n_truncate)
```

```
+-----+
|                                     message|
+-----+
|[INFO]: 2022-09-05 03:35:01.43 LOOKING FOR WORK...|
|[WARN]: 2022-09-05 03:35:58.007 WORKERS ARE UNA...|
|[INFO]: 2022-09-05 03:40:59.054 LOOKING FOR WOR...|
|[INFO]: 2022-09-05 03:42:24 3 WORKERS WERE ACQU...|
|[INFO]: 2022-09-05 03:42:37 INITIALIZING INSTAN...|
+-----+
only showing top 5 rows
```

```
# Change to title case
# (first letter of each word is upper case):
m.withColumn('message', initcap('message'))\
  .show(5, truncate = n_truncate)
```

```
+-----+
|                                     message|
+-----+
|[info]: 2022-09-05 03:35:01.43 Looking For Work...|
|[warn]: 2022-09-05 03:35:58.007 Workers Are Una...|
|[info]: 2022-09-05 03:40:59.054 Looking For Wor...|
|[info]: 2022-09-05 03:42:24 3 Workers Were Acqu...|
|[info]: 2022-09-05 03:42:37 Initializing Instan...|
+-----+
only showing top 5 rows
```

## 10.3 Calculating string length

In Spark, you can use the `length()` function to get the length (i.e. the number of characters) of a string. In the example below, we can see that the first log message is 74 characters long, while the second log message have 112 characters.

```
from pyspark.sql.functions import length
logs\
  .withColumn('length', length('message'))\
  .show(5)
```

```
+-----+-----+-----+
|          ip|          message|length|
+-----+-----+-----+
| 1.0.104.27 |[INFO]: 2022-09-0...|    74|
| 1.0.104.27 |[WARN]: 2022-09-0...|   112|
| 1.0.104.27 |[INFO]: 2022-09-0...|    75|
| 1.0.104.27 |[INFO]: 2022-09-0...|    94|
| 1.0.104.27 |[INFO]: 2022-09-0...|    65|
+-----+-----+-----+
```

only showing top 5 rows

## 10.4 Trimming or removing spaces from strings

The process of removing unnecessary spaces from strings is usually called “trimming”. In Spark, we have three functions that do this process, which are:

- `trim()`: removes spaces from both sides of the string;
- `ltrim()`: removes spaces from the left side of the string;
- `rtrim()`: removes spaces from the right side of the string;

```
from pyspark.sql.functions import (
    trim, rtrim, ltrim
)

logs\
  .select('ip')\
  .withColumn('ip_trim', trim('ip'))\
  .withColumn('ip_ltrim', ltrim('ip'))\
  .withColumn('ip_rtrim', rtrim('ip'))\
```

```
.show(5)
```

```
+-----+-----+-----+-----+
|          ip|  ip_trim|  ip_ltrim|  ip_rtrim|
+-----+-----+-----+-----+
|  1.0.104.27 |1.0.104.27|1.0.104.27 | 1.0.104.27|
|  1.0.104.27 |1.0.104.27|1.0.104.27 | 1.0.104.27|
|  1.0.104.27 |1.0.104.27|1.0.104.27 | 1.0.104.27|
|  1.0.104.27 |1.0.104.27|1.0.104.27 | 1.0.104.27|
|  1.0.104.27 |1.0.104.27|1.0.104.27 | 1.0.104.27|
+-----+-----+-----+-----+
```

only showing top 5 rows

For the most part, I tend to remove these unnecessary strings when I want to: 1) tidy the values; 2) avoid weird and confusing mistakes in filters on my DataFrame. The second case is worth describing in more details.

Let's suppose you wanted to filter all rows from the logs DataFrame where ip is equal to the 1.0.104.27 IP address. However, you can see in the result above, that I get nothing. Not a single row of result.

```
from pyspark.sql.functions import col
logs.filter(col('ip') == "1.0.104.27")\
    .show(5)
```

```
+---+-----+
| ip|message|
+---+-----+
+---+-----+
```

But if you see the result of the previous example (where we applied the three versions of “trim functions”), you know that this IP address 1.0.104.27 exists in the DataFrame. You know that the filter above should find values for this IP address. So why it did not find any rows?

The answer is these annoying (and hidden) spaces on both sides of the values from the ip column. If we remove these unnecessary spaces from the values of the ip column, we suddenly find the rows that we were looking for.

```
logs.filter(trim(col('ip')) == "1.0.104.27")\
    .show(5)
```

```

+-----+-----+
|          ip|          message|
+-----+-----+
| 1.0.104.27 |[INFO]: 2022-09-0...|
| 1.0.104.27 |[WARN]: 2022-09-0...|
| 1.0.104.27 |[INFO]: 2022-09-0...|
| 1.0.104.27 |[INFO]: 2022-09-0...|
| 1.0.104.27 |[INFO]: 2022-09-0...|
+-----+-----+
only showing top 5 rows

```

## 10.5 Extracting substrings

There are five main functions that we can use in order to extract substrings of a string, which are:

- `substring()` and `substr()`: extract a single substring based on a start position and the length (number of characters) of the collected substring<sup>2</sup>;
- `substring_index()`: extract a single substring based on a delimiter character<sup>3</sup>;
- `split()`: extract one or multiple substrings based on a delimiter character;
- `regexp_extract()`: extracts substrings from a given string that match a specified regular expression pattern;

You can obviously extract a substring that matches a particular regex (regular expression) as well, by using the `regexp_extract()` function. However, I will describe this function, and the regex functionality available in pyspark at Section 10.7, or, more specifically, at Section 10.7.5. For now, just understand that you can also use regex to extract substrings from your text data.

### 10.5.1 A substring based on a start position and length

The `substring()` and `substr()` functions they both work the same way. However, they come from different places. The `substring()` function comes from the `spark.sql.functions` module, while the `substr()` function is actually a method from the `Column` class.

One interesting aspect of these functions, is that they both use a one-based index, instead of a zero-based index. This means that the first character in the full string is identified by the index 1, instead of the index 0.

The first argument in both function is the index that identifies the start position of the substring. If you set this argument to, let's say, 4, it means that the substring you want to extract starts at the 4th character in the input string.

<sup>2</sup>Instead of using a zero based index (which is the default for Python), these functions use a one based index.

<sup>3</sup>Instead of using a zero based index (which is the default for Python), these functions use a one based index.

The second argument is the amount of characters in the substring, or, in other words, it's length. For example, if you set this argument to 10, it means that the function will extract the substring that is formed by walking  $10 - 1 = 9$  characters ahead from the start position you specified at the first argument. We can also interpret this as: the function will walk ahead on the string, from the start position, until it gets a substring that is 10 characters long.

In the example below, we are extracting the substring that starts at the second character (index 2) and ends at the sixth character (index 6) in the string.

```
from pyspark.sql.functions import col, substring
# `df1` and `df2` are equal, because
# they both mean the same thing
df1 = (logs
      .withColumn('sub', col('message').substr(2, 5))
      )

df2 = (logs
      .withColumn('sub', substring('message', 2, 5))
      )

df2.show(5)
```

```
+-----+-----+-----+
|          ip|          message|  sub|
+-----+-----+-----+
|  1.0.104.27 | [INFO]: 2022-09-0... | INFO|
|  1.0.104.27 | [WARN]: 2022-09-0... | WARN|
|  1.0.104.27 | [INFO]: 2022-09-0... | INFO|
|  1.0.104.27 | [INFO]: 2022-09-0... | INFO|
|  1.0.104.27 | [INFO]: 2022-09-0... | INFO|
+-----+-----+-----+
only showing top 5 rows
```

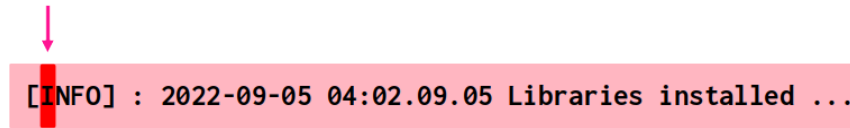
Just to be very clear on how `substring()` and `substr()` both works. The Figure [10.1](#) illustrates the result of the above code.

## 10.5.2 A substring based on a delimiter

The `substring_index()` function works very differently. It collects the substring formed between the start of the string, and the *n*th occurrence of a particular character.

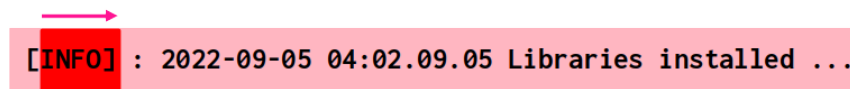


Start here at the 2nd character



[INFO] : 2022-09-05 04:02.09.05 Libraries installed ...

Then, move forward until we have a 5 characters long substring



[INFO] : 2022-09-05 04:02.09.05 Libraries installed ...

Figure 10.1: How substring() and substr() works

For example, if you ask `substring_index()` to search for the 3rd occurrence of the character `$` in your string, the function will return to you the substring formed by all characters that are between the start of the string until the 3rd occurrence of this character `$`.

You can also ask `substring_index()` to read backwards. That is, to start the search on the end of the string, and move backwards in the string until it gets to the 3rd occurrence of this character `$`.

As an example, let's look at the 10th log message present in the `logs` DataFrame. I used the `collect()` DataFrame method to collect this message into a raw python string, so we can easily see the full content of the message.

```
from pyspark.sql.functions import monotonically_increasing_id

mes_10th = (
    logs
    .withColumn(
        'row_id',
        monotonically_increasing_id()
    )
    .where(col('row_id') == 9)
)

message = mes_10th.collect()[0]['message']
print(message)
```

```
[INFO]: 2022-09-05 04:02:09.05 Libraries installed: pandas, flask, numpy,
spark_map, pyspark
```

We can see that this log message is listing a set of libraries that were installed somewhere. Suppose you want to collect the first and the last libraries in this list. How would you do it?

A good start is to isolate the list of libraries from the rest of the message. In other words, there is a bunch of characters in the start of the log message, that we do not care about. So let's get rid of them.

If you look closely to the message, you can see that the character `:` appears twice within the message. One close to the start of the string, and another time right before the start of the list of the libraries. We can use this character as our first delimiter, to collect the third substring that it creates within the total string, which is the substring that contains the list of libraries.

This first stage is presented visually at Figure 10.2.

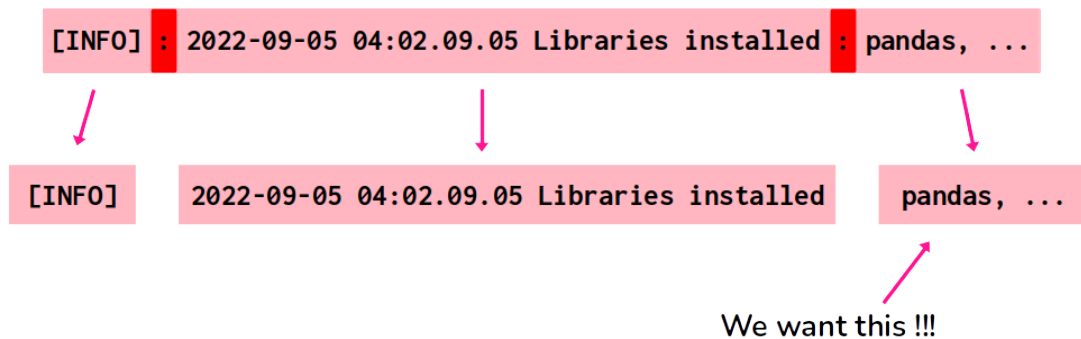


Figure 10.2: Substrings produced by the `:` delimiter character

Now that we identified the substrings produced by the “delimiter character”, we just need to understand better which index we need to use in `substring_index()` to get this third substring that we want. The Figure 10.3 presents in a visual manner how the count system of `substring_index()` works.

When you use a positive index, `substring_index()` will count the occurrences of the delimiter character from left to right. But, when you use a negative index, the opposite happens. That is, `substring_index()` counts the occurrences of the delimiter character from right to left.

The index 1 represents the first substring that is before the 1st occurrence of the delimiter (`[INFO]`). The index 2 represents everything that is before the 2nd occurrence of the delimiter (`[INFO]: 2022-09-05 04:02.09.05 Libraries installed`). etc.

In contrast, the index -1 represents everything that is after the 1st occurrence of the delimiter, counting from right to left (`pandas, flask, numpy, spark_map, pyspark`). The index -2 represents everything that is after the 2nd occurrence of the delimiter (`2022-09-05 04:02.09.05 Libraries installed: pandas, flask, numpy, spark_map, pyspark`). Again, counting from right to left.

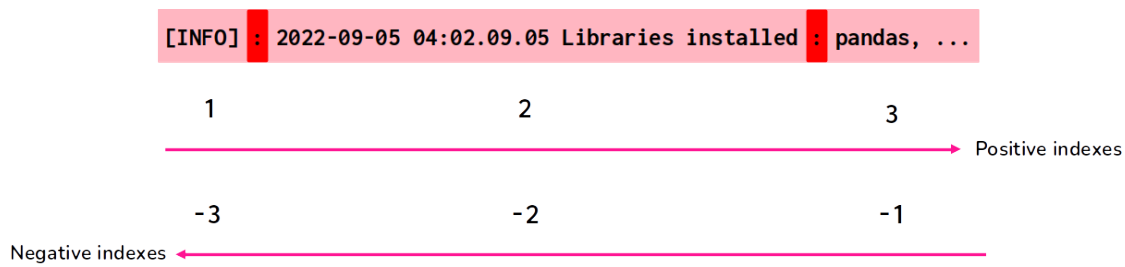


Figure 10.3: The count system of substring\_index()

Having all these informations in mind, we can conclude that the following code fit our first objective. Note that I applied the trim() function over the result of substring\_index(), to ensure that the result substring does not contain any unnecessary spaces at both ends.

```
from pyspark.sql.functions import substring_index
mes_10th = mes_10th\
    .withColumn(
        'list_of_libraries',
        trim(substring_index('message', ':', -1))
    )

mes_10th.select('list_of_libraries')\
    .show(truncate = n_truncate)
```

```
+-----+
|               list_of_libraries|
+-----+
|pandas, flask, numpy, spark_map, pyspark|
+-----+
```

### 10.5.3 Forming an array of substrings

Now is a good time to introduce the split() function, because we can use it to extract the first and the last library from the list libraries of stored at the mes\_10th DataFrame.

Basically, this function also uses a delimiter character to cut the total string into multiple pieces. However, this function stores these multiple pieces (or multiple substrings) into an array of substrings. With this strategy, we can now access each substring (or each piece of the total string) individually.

If we look again at the string that we stored at the `list_of_libraries` column, we have a list of libraries, separated by a comma.

```
mes_10th\  
  .select('list_of_libraries')\  
  .show(truncate = n_truncate)
```

```
+-----+  
|               list_of_libraries|  
+-----+  
|pandas, flask, numpy, spark_map, pyspark|  
+-----+
```

The comma character (,) plays an important role in this string, by separating each value in the list. And we can use this comma character as the delimiter inside `split()`, to get an array of substrings. Each element of this array is one of the many libraries in the list. The Figure 10.4 presents this process visually.

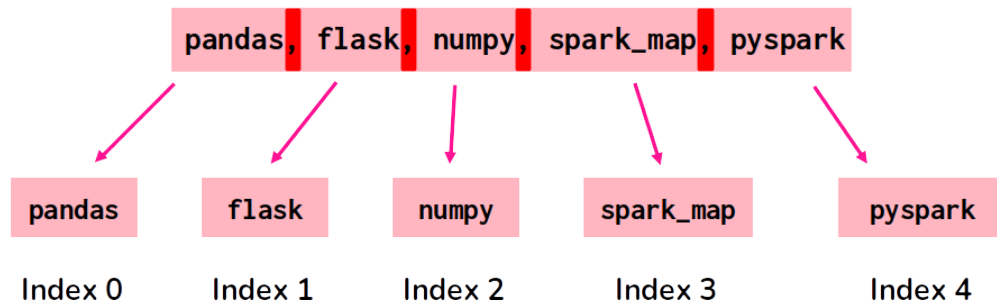


Figure 10.4: Building an array of substrings with `split()`

The code to make this process is very straightforward. In the example below, the column `array_of_libraries` becomes a column of data type `ArrayType(StringType)`, that is, an array of string values.

```
from pyspark.sql.functions import split  
mes_10th = mes_10th\  
  .withColumn(  
    'array_of_libraries',  
    split('list_of_libraries', ', ')
```

```

    )

mes_10th\
    .select('array_of_libraries')\
    .show(truncate = n_truncate)

```

```

+-----+
|               array_of_libraries|
+-----+
|[pandas, flask, numpy, spark_map, pyspark]|
+-----+

```

By having this array of substring, we can very easily select a specific element in this array, by using the `getItem()` column method, or, by using the open brackets as you would normally use to select an element in a python list.

You just need to give the index of the element you want to select, like in the example below that we select the first and the fifth libraries in the array.

```

mes_10th\
    .withColumn('lib_1', col('array_of_libraries')[0])\
    .withColumn('lib_5', col('array_of_libraries').getItem(4))\
    .select('lib_1', 'lib_5')\
    .show(truncate = n_truncate)

```

```

+-----+-----+
| lib_1|  lib_5|
+-----+-----+
|pandas|pyspark|
+-----+-----+

```

## 10.6 Concatenating multiple strings together

Sometimes, we need to concatenate multiple strings together, to form a single and longer string. To do this process, Spark offers two main functions, which are: `concat()` and `concat_ws()`. Both of these functions receives a list of columns as input, and will perform the same task, which is to concatenate the values of each column in the list, sequentially.

However, the `concat_ws()` function have an extra argument called `sep`, where you can define a string to be used as the separator (or the “delimiter”) between the values of each column in the list. In some

way, this sep argument and the concat\_ws() function works very similarly to the [join\(\) string method of python](#)<sup>4</sup>.

Let's comeback to the penguins DataFrame to demonstrate the use of these functions:

```
path = "../Data/penguins.csv"
penguins = spark.read\
    .csv(path, header = True)

penguins.select('species', 'island', 'sex')\
    .show(5)
```

```
+-----+-----+-----+
|species|  island|   sex|
+-----+-----+-----+
| Adelie|Torgersen|  male|
| Adelie|Torgersen|female|
| Adelie|Torgersen|female|
| Adelie|Torgersen|  NULL|
| Adelie|Torgersen|female|
+-----+-----+-----+
only showing top 5 rows
```

Suppose you wanted to concatenate the values of the columns species, island and sex together, and, store these new values on a separate column. All you need to do is to list these columns inside the concat() or concat\_ws() function.

If you look at the example below, you can see that I also used the lit() function to add a underline character (\_) between the values of each column. This is more verbose, because if you needed to concatenate 10 columns together, and still add a “delimiter character” (like the underline) between the values of each column, you would have to write lit('\_') for 9 times on the list.

In contrast, the concat\_ws() offers a much more succinct way of expressing this same operation. Because the first argument of concat\_ws() is the character to be used as the delimiter between each column, and, after that, we have the list of columns to be concatenated.

```
from pyspark.sql.functions import (
    concat,
    concat_ws,
    lit
)
```

---

<sup>4</sup><https://docs.python.org/3/library/stdtypes.html#str.join>

```

penguins\
  .withColumn(
    'using_concat',
    concat(
      'species', lit('_'), 'island',
      lit('_'), 'sex')
  )\
  .withColumn(
    'using_concat_ws',
    concat_ws(
      '_', # The delimiter character
      'species', 'island', 'sex' # The list of columns
    )
  )\
  .select('using_concat', 'using_concat_ws')\
  .show(5, truncate = n_truncate)

```

```

+-----+-----+
|      using_concat|      using_concat_ws|
+-----+-----+
| Adeline_Torgersen_male| Adeline_Torgersen_male|
| Adeline_Torgersen_female| Adeline_Torgersen_female|
| Adeline_Torgersen_female| Adeline_Torgersen_female|
|              NULL|      Adeline_Torgersen|
| Adeline_Torgersen_female| Adeline_Torgersen_female|
+-----+-----+
only showing top 5 rows

```

If you look closely to the result above, you can also see, that `concat()` and `concat_ws()` functions deal with null values in different ways. If `concat()` finds a null value for a particular row, in any of the listed columns to be concatenated, the end result of the process is a null value for that particular row.

On the other hand, `concat_ws()` will try to concatenate as many values as he can. If he does find a null value, he just ignores this null value and go on to the next column, until it hits the last column in the list.

## 10.7 Introducing regular expressions

Spark also provides some basic regex (*regular expressions*) functionality. Most of this functionality is available through two functions that comes from the `pyspark.sql.functions` module, which are:

- `regexp_replace()`: replaces all occurrences of a specified regular expression pattern in a given string with a replacement string.;
- `regexp_extract()`: extracts substrings from a given string that match a specified regular expression pattern;

There is also a column method that provides an useful way of testing if the values of a column matches a regular expression or not, which is the `rlike()` column method. You can use the `rlike()` method in conjunction with the `filter()` or `where()` DataFrame methods, to find all values that fit (or match) a particular regular expression, like we demonstrated at Section [5.5.7.2](#).

### 10.7.1 The Java regular expression standard

At this point, is worth remembering a basic fact about Apache Spark that we introduced at Chapter 2. Apache Spark is written in Scala, which is a modern programming language deeply connected with the Java programming language. One of the many consequences from this fact, is that all regular expression functionality available in Apache Spark is based on the Java `java.util.regex` package.

This means that you should always write regular expressions on your pyspark code that follows the Java regular expression syntax, and not the Python regular expression syntax, which is based on the python module `re`.

Although this detail is important, these two flavors of regular expressions (Python syntax versus Java syntax) are very, very similar. So, for the most part, you should not see any difference between these two syntaxes.

If for some reason, you need to consult the full list of all metacharacters available in the Java regular expression standard, you can always check the Java documentation for the `java.util.regex` package. More specifically, the [documentation for the `java.util.regex.Pattern` class](#)<sup>5</sup>.

The following list gives you a quick description of a small fraction of the available metacharacters in the Java syntax, and, as a result, metacharacters that you can use in pyspark:

- `.` : Matches any single character;
- `*` : Matches zero or more occurrences of the preceding character or pattern;
- `+` : Matches one or more occurrences of the preceding character or pattern;
- `?` : Matches zero or one occurrence of the preceding character or pattern;
- `|` : Matches either the expression before or after the `|`;
- `[]` : Matches any single character within the brackets;
- `\d` : Matches any digit character;
- `\b` : Matches a word boundary character;
- `\w` : Matches any word character. Equivalent to the `"\b([a-zA-Z_0-9]+)\b"` regular expression;
- `\s` : Matches any whitespace character;
- `()` : Groups a series of pattern elements to a single element;

---

<sup>5</sup><https://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html>



## 10.7.2 Using an invalid regular expression

When you write an invalid regular expression in your code, Spark usually complains with a `java.util.regex.PatternSyntaxException` runtime error. The code presented below is an example of code that produces such error.

In this example, the regular expression `\b([a-z]` is invalid because it is missing a closing parenthesis. If you try to execute this code, Spark will raise a with the message “Unclosed group near index 7”. This error message indicates that there is a syntax error in the regular expression, due to an unclosed group (i.e., a missing closing parenthesis).

```
from pyspark.sql.functions import col
weird_regex = '\b([a-z]'
logs\
    .filter(col('message').rlike(weird_regex))\
    .show(5)
```

```
Py4JJavaError: An error occurred while calling o261.showString.
:   java.util.regex.PatternSyntaxException: Unclosed group near index 7
([a-z]
```

To avoid these runtime errors, due to invalid regular expressions, is always a good idea to test your regular expressions, before you use them in your pyspark code. You can easily test your regular expressions by using online tools, such as the [Regex101 website](https://regex101.com/)<sup>6</sup>.

## 10.7.3 Replacing occurrences of a particular regular expression with `regexp_replace()`

One of the most essential actions with regular expression is to find text that fits into a particular regular expression, and, rewriting this text into a different format, or, even removing it completely from the string.

The `regexp_replace()` function (from the `pyspark.sql.functions` module) is the function that allows you to perform this kind of operation on string values of a column in a Spark DataFrame.

This function replaces all occurrences of a specified regular expression pattern in a given string with a replacement string, and it takes three different arguments:

- The input column name or expression that contains the string values to be modified;
- The regular expression pattern to search for within the input string values;

---

<sup>6</sup><https://regex101.com/>

- The replacement string that will replace all occurrences of the matched pattern in the input string values;

As an example, let's suppose we want to remove completely the type of the message in all log messages present in the logs DataFrame. To that, we first need to get a regular expression capable of identifying all possibilities for these types.

A potential candidate would be the regular expression '\\[(INFO|ERROR|WARN)\\]: ', so let's give it a shot. Since we are trying to **remove** this particular part from all log messages, we should replace this part of the string by an empty string (''), like in the example below:

```
from pyspark.sql.functions import regexp_replace

type_regex = '\\[(INFO|ERROR|WARN)\\]: '

logs\
  .withColumn(
    'without_type',
    regexp_replace('message', type_regex, '')
  )\
  .select('message', 'without_type')\
  .show(truncate = 30)
```

message	without_type
[INFO]: 2022-09-05 03:35:01...   2022-09-05 03:35:01.43 Look...	2022-09-05 03:35:01.43 Look...
[WARN]: 2022-09-05 03:35:58...   2022-09-05 03:35:58.007 Wor...	2022-09-05 03:35:58.007 Wor...
[INFO]: 2022-09-05 03:40:59...   2022-09-05 03:40:59.054 Loo...	2022-09-05 03:40:59.054 Loo...
[INFO]: 2022-09-05 03:42:24...   2022-09-05 03:42:24 3 Worke...	2022-09-05 03:42:24 3 Worke...
[INFO]: 2022-09-05 03:42:37...   2022-09-05 03:42:37 Initial...	2022-09-05 03:42:37 Initial...
[WARN]: 2022-09-05 03:52:02...   2022-09-05 03:52:02.98 Libr...	2022-09-05 03:52:02.98 Libr...
[INFO]: 2022-09-05 04:00:33...   2022-09-05 04:00:33.210 Lib...	2022-09-05 04:00:33.210 Lib...
[INFO]: 2022-09-05 04:01:15...   2022-09-05 04:01:15 All clu...	2022-09-05 04:01:15 All clu...
[INFO]: 2022-09-05 04:01:35...   2022-09-05 04:01:35.022 Mak...	2022-09-05 04:01:35.022 Mak...
[INFO]: 2022-09-05 04:02:09...   2022-09-05 04:02:09.05 Libr...	2022-09-05 04:02:09.05 Libr...
[INFO]: 2022-09-05 04:02:09...   2022-09-05 04:02:09.05 The ...	2022-09-05 04:02:09.05 The ...
[INFO]: 2022-09-05 04:02:09...   2022-09-05 04:02:09.05 An e...	2022-09-05 04:02:09.05 An e...
[ERROR]: 2022-09-05 04:02:1...   2022-09-05 04:02:12 A task ...	2022-09-05 04:02:12 A task ...
[ERROR]: 2022-09-05 04:02:3...   2022-09-05 04:02:34.111 Err...	2022-09-05 04:02:34.111 Err...
[ERROR]: 2022-09-05 04:02:3...   2022-09-05 04:02:34.678 Tra...	2022-09-05 04:02:34.678 Tra...
[ERROR]: 2022-09-05 04:02:3...   2022-09-05 04:02:35.14 Quit...	2022-09-05 04:02:35.14 Quit...

Is useful to remind that this `regexp_replace()` function searches for **all occurrences** of the regular expression on the input string values, and replaces all of these occurrences by the input replacement string that you gave. However, if the function does not find any matches for your regular expression inside a particular value in the column, then, the function simply returns this value intact.

#### 10.7.4 Introducing capturing groups on pyspark

One of the many awesome functionalities of regular expressions, is the capability of enclosing parts of a regular expression inside groups, and actually store (or cache) the substring matched by this group. This process of grouping parts of a regular expression inside a group, and capturing substrings with them, is usually called of “grouping and capturing”.

Is worth pointing out that this capturing groups functionality is available both in `regexp_replace()` and `regexp_extract()`.

##### 10.7.4.1 What is a capturing group ?

Ok, but, what is this group thing? You create a group inside a regular expression by enclosing a particular section of your regular expression inside a pair of parentheses. The regular expression that is written inside this pair of parentheses represents a capturing group.

A capturing group inside a regular expression is used to capture a specific part of the matched string. This means that the actual part of the input string that is matched by the regular expression that is inside this pair of parentheses, is captured (or cached, or saved) by the group, and, can be reused later.

Besides grouping part of a regular expression together, parentheses also create a numbered capturing group. It stores the part of the string matched by the part of the regular expression inside the parentheses. .... The regex “Set(Value)?” matches “Set” or “SetValue”. In the first case, the first (and only) capturing group remains empty. In the second case, the first capturing group matches “Value”. (Goyvaerts 2023).

So, remember, to use capturing groups in a regular expression, you must enclose the part of the pattern that you want to capture in parentheses (). Each set of parentheses creates a new capturing group. This means that you can create multiple groups inside a single regular expression, and, then, reuse latter the substrings captured by all of these multiple groups. Awesome, right?

Each new group (that is, each pair of parentheses) that you create in your regular expression have a different index. That means that the first group is identified by the index 1, the second group, by the index 2, the third group, by the index 3, etc.

Just to quickly demonstrate these capturing groups, here is a quick example, in pure Python:

```
import re

# A regular expression that contains
# three different capturing groups
regex = r"(\d{3})-(\d{2})-(\d{4})"

# Match the regular expression against a string
text = "My social security number is 123-45-6789."
match = re.search(regex, text)

# Access the captured groups
group1 = match.group(1) # "123"
group2 = match.group(2) # "45"
group3 = match.group(3) # "6789"
```

In the above example, the regular expression `r"(\d{3})-(\d{2})-(\d{4})"` contains three capturing groups, each enclosed in parentheses. When the regular expression is matched against the string "My social security number is 123-45-6789.", the first capturing group matches the substring "123", the second capturing group matches "45", and the third capturing group matches "6789".

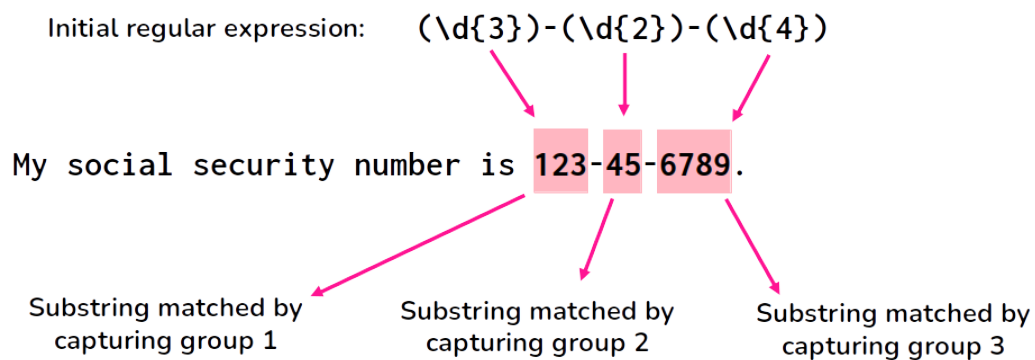


Figure 10.5: Example of capturing groups

In Python, we can access the captured groups using the `group()` method of the Match object returned by `re.search()`. In this example, `match.group(1)` returns the captured substring of the first capturing group (which is "123"), `match.group(2)` returns second "45", and `match.group(3)` returns "6789".

#### 10.7.4.2 How can we use capturing groups in pyspark ?

Ok, now that we understood what capturing groups is, how can we use them in pyspark? First, remember, capturing groups will be available to you, only if you enclose a part of your regular expression

in a pair of parentheses. So the first part is to make sure that the capturing groups are present in your regular expressions.

After that, you can access the substring matched by the capturing group, by using the reference index that identifies this capturing group you want to use. In pure Python, we used the `group()` method with the group index (like 1, 2, etc.) to access these values.

But in pyspark, we access these groups by using a special pattern formed by the group index preceded by a dollar sign (\$). That is, the text `$1` references the first capturing group, `$2` references the second capturing group, etc.

As a first example, lets go back to the regular expression we used at Section 10.7.3: `\\[(INFO|ERROR|WARN)\\]:.`. This regular expression contains one capturing group, which captures the type label of the log message: `(INFO|ERROR|WARN)`.

If we use the special pattern `$1` to reference this capturing group inside of `regexp_replace()`, what is going to happen is: `regexp_replace()` will replace all occurrences of the input regular expression found on the input string, by the substring matched by the first capturing group. See in the example below:

```
logs\
  .withColumn(
    'using_groups',
    regexp_replace('message', type_regex, 'Type Label -> $1 | ')
  )\
  .select('message', 'using_groups')\
  .show(truncate = 30)
```

message	using_groups
[INFO]: 2022-09-05 03:35:01...	Type Label -> INFO   2022-0...
[WARN]: 2022-09-05 03:35:58...	Type Label -> WARN   2022-0...
[INFO]: 2022-09-05 03:40:59...	Type Label -> INFO   2022-0...
[INFO]: 2022-09-05 03:42:24...	Type Label -> INFO   2022-0...
[INFO]: 2022-09-05 03:42:37...	Type Label -> INFO   2022-0...
[WARN]: 2022-09-05 03:52:02...	Type Label -> WARN   2022-0...
[INFO]: 2022-09-05 04:00:33...	Type Label -> INFO   2022-0...
[INFO]: 2022-09-05 04:01:15...	Type Label -> INFO   2022-0...
[INFO]: 2022-09-05 04:01:35...	Type Label -> INFO   2022-0...
[INFO]: 2022-09-05 04:02:09...	Type Label -> INFO   2022-0...
[INFO]: 2022-09-05 04:02:09...	Type Label -> INFO   2022-0...
[INFO]: 2022-09-05 04:02:09...	Type Label -> INFO   2022-0...
[ERROR]: 2022-09-05 04:02:1...	Type Label -> ERROR   2022-...

```
|[ERROR]: 2022-09-05 04:02:3...|Type Label -> ERROR | 2022-...|
|[ERROR]: 2022-09-05 04:02:3...|Type Label -> ERROR | 2022-...|
|[ERROR]: 2022-09-05 04:02:3...|Type Label -> ERROR | 2022-...|
+-----+-----+
```

In essence, you can reuse the substrings matched by the capturing groups, by using the special patterns \$1, \$2, \$3, etc. This means that you can reuse the substrings captured by multiple groups at the same time inside `regexp_replace()` and `regexp_extract()`. For example, if we use the replacement string "\$1, \$2, \$3" inside `regexp_replace()`, we would get the substrings matched by the first, second and third capturing groups, separated by commas.

However, it is also good to emphasize a small limitation that this system has. When you need to reuse the substrings captured by multiple groups together, it is important that you make sure to add some amount of space (or some delimiter character) between each group reference, like "\$1 \$2 \$3".

Because if you write these group references one close to each other (like in "\$1\$2\$3"), it is not going to work. In other words, Spark will not understand that you are trying to access a capturing group. It will interpret the text "\$1\$2\$3" as the literal value "\$1\$2\$3", and not as a special pattern that references multiple capturing groups in the regular expression.

### 10.7.5 Extracting substrings with `regexp_extract()`

Another very useful regular expression activity is to extract a substring from a given string that match a specified regular expression pattern. The `regexp_extract()` function is the main method used to do this process.

This function takes three arguments, which are:

- The input column name or expression that contains the string to be searched;
- The regular expression pattern to search for within the input string;
- The index of the capturing group within the regular expression pattern that corresponds to the substring to extract;

You may (or may not) use capturing groups inside of `regexp_replace()`. However, on the other hand, the `regexp_extract()` function **is based on** the capturing groups functionality. As a consequence, when you use `regexp_extract()`, you must give a regular expression that **contains some capturing group**. Because otherwise, the `regexp_extract()` function becomes useless.

In other words, the `regexp_extract()` function extracts substrings that are matched by the capturing groups present in your input regular expression. If you want, for example, to use `regexp_extract()` to extract the substring matched by a entire regular expression, then, you just need to surround this entire regular expression by a pair of parentheses. This way you transform this entire regular expression in a capturing group, and, therefore, you can extract the substring matched by this group.

As an example, let's go back again to the regular expression we used in the logs DataFrame: `\\[(INFO|ERROR|WARN)\\]:`. We can extract the type of log message label, by using the index 1 to reference the first (and only) capturing group in this regular expression.

```
from pyspark.sql.functions import regexp_extract

logs\
  .withColumn(
    'message_type',
    regexp_extract('message', type_regex, 1)
  )\
  .select('message', 'message_type')\
  .show(truncate = 30)
```

message	message_type
[INFO]: 2022-09-05 03:35:01...	INFO
[WARN]: 2022-09-05 03:35:58...	WARN
[INFO]: 2022-09-05 03:40:59...	INFO
[INFO]: 2022-09-05 03:42:24...	INFO
[INFO]: 2022-09-05 03:42:37...	INFO
[WARN]: 2022-09-05 03:52:02...	WARN
[INFO]: 2022-09-05 04:00:33...	INFO
[INFO]: 2022-09-05 04:01:15...	INFO
[INFO]: 2022-09-05 04:01:35...	INFO
[INFO]: 2022-09-05 04:02:09...	INFO
[INFO]: 2022-09-05 04:02:09...	INFO
[INFO]: 2022-09-05 04:02:09...	INFO
[ERROR]: 2022-09-05 04:02:1...	ERROR
[ERROR]: 2022-09-05 04:02:3...	ERROR
[ERROR]: 2022-09-05 04:02:3...	ERROR
[ERROR]: 2022-09-05 04:02:3...	ERROR

As another example, let's suppose we wanted to extract not only the type of the log message, but also, the timestamp and the content of the message, and store these different elements in separate columns.

To do that, we could build a more complete regular expression. An expression capable of matching the entire log message, and, at the same time, capture each of these different elements inside a different capturing group. The code below is an example that produces such regular expression, and applies it over the logs DataFrame.

```

type_regex = r'\[(INFO|ERROR|WARN)\]: '

date_regex = r'\d{4}-\d{2}-\d{2}'
time_regex = r' \d{2}:\d{2}:\d{2}([.]\d+)?'
timestamp_regex = date_regex + time_regex
timestamp_regex = r'(' + timestamp_regex + r')'

regex = type_regex + timestamp_regex + r'(.+)$'

logs\
  .withColumn(
    'message_type',
    regexp_extract('message', regex, 1)
  )\
  .withColumn(
    'timestamp',
    regexp_extract('message', regex, 2)
  )\
  .withColumn(
    'message_content',
    regexp_extract('message', regex, 4)
  )\
  .select('message_type', 'timestamp', 'message_content')\
  .show(5, truncate = 30)

```

```

+-----+-----+-----+
|message_type|          timestamp|          message_content|
+-----+-----+-----+
|      INFO| 2022-09-05 03:35:01.43| Looking for workers at Sou...|
|      WARN|2022-09-05 03:35:58.007| Workers are unavailable at...|
|      INFO|2022-09-05 03:40:59.054| Looking for workers at Sou...|
|      INFO|   2022-09-05 03:42:24| 3 Workers were acquired at...|
|      INFO|   2022-09-05 03:42:37| Initializing instances in ...|
+-----+-----+-----+

```

only showing top 5 rows

### 10.7.6 Identifying values that match a particular regular expression with `rlike()`

The `rlike()` column method is useful for checking if a string value in a column matches a specific regular expression. We briefly introduced this method at Section [5.5.7.2](#). This method has only one input, which is the regular expression you want to apply over the column values.



As an example, let's suppose you wanted to identify timestamp values inside your strings. You could use a regular expression pattern to find which text values had these kinds of values inside them.

A possible regular expression candidate would be "[0-9]{2}:[0-9]{2}:[0-9]{2}([.][0-9]+)?". This regex matches timestamp values in the format "hh:mm:ss.sss". This pattern consists of the following building blocks, or, elements:

- [0-9]{2}: Matches any two digits from 0 to 9.
- :: Matches a colon character.
- ([.][0-9]+)?: Matches an optional decimal point followed by one or more digits.

If we apply this pattern over all log messages stored in the logs DataFrame, we would find that all log messages matches this particular regular expression. Because all log messages contains a timestamp value at the start of the message:

```
from pyspark.sql.functions import col

pattern = "[0-9]{2}:[0-9]{2}:[0-9]{2}([.][0-9]+)?"
logs\
  .withColumn(
    'does_it_match?',
    col("message").rlike(pattern)
  )\
  .select('message', 'does_it_match?')\
  .show(5, truncate = n_truncate)
```

```
+-----+-----+
|                                     message|does_it_match?|
+-----+-----+
|[INFO]: 2022-09-05 03:35:01.43 Looking for work...|      true|
|[WARN]: 2022-09-05 03:35:58.007 Workers are una...|      true|
|[INFO]: 2022-09-05 03:40:59.054 Looking for wor...|      true|
|[INFO]: 2022-09-05 03:42:24 3 Workers were acqu...|      true|
|[INFO]: 2022-09-05 03:42:37 Initializing instan...|      true|
+-----+-----+
```

only showing top 5 rows

# 11 Tools for dates and datetimes manipulation

Units of measurement that represent time are very common types of data in our modern world. Nowadays, dates and datetimes (or timestamps) are the most common units used to represent a specific point in time. In this chapter, you will learn how to import, manipulate and use this kind of data with pyspark.

In Spark, dates and datetimes are represented by the `DateType` and `TimestampType` data types, respectively, which are available in the `pyspark.sql.types` module. Spark also offers two other data types to represent “intervals of time”, which are `YearMonthIntervalType` and `DayTimeIntervalType`. However, you usually don’t use these types directly to create new objects. In other words, they are intermediate types. They are a passage, or a path you use to get to another data type.

## 11.1 Creating date values

Dates are normally interpreted in pyspark using the `DateType` data type. There are three common ways to create date objects, which are:

1. from strings (like "3 of June of 2023", or maybe, "2023-02-05").
2. by extracting the date component from datetime values (i.e. values of type `TimestampType`).
3. by combining day, month and year components to build a date object.

### 11.1.1 From strings

When you have a `StringType` column in your `DataFrame` that contains dates that are currently being stored inside strings, and you want to convert this column into a `DateType` column, you basically have two choices: 1) use the automatic column conversion with `cast()` or `astype()`; 2) use the `to_date()` Spark SQL function to convert the strings using a specific date format.

When you use the `cast()` (or `astype()`) column method that we introduced at Section 5.7.3, Spark will perform a quick and automatic conversion to `DateType` by casting the strings you have into the `DateType`. But when you use this method, Spark will always assume that the dates you have are in the ISO-8601 format, which is the international standard for dates. This format is presented at Figure 11.1:

## ISO-8601 format for dates

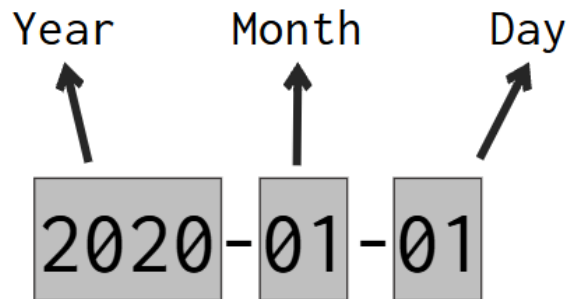


Figure 11.1: The ISO-8601 format for dates

Basically, the ISO-8601 standard specifies that dates are represented in the format “YYYY-MM-DD” (or “Year-Month-Date”), like 2023-09-19, 1926-05-21, or 2005-11-01. This is also the format that dates are usually formatted in United States of America. So, if the dates you have (which are currently stored inside strings) are formatted like the ISO-8601 standard, then, you can safely and easily convert them into the `DateTime` by using the `cast()` or `astype()` column methods.

However, if these dates are formatted in a different way, then, the `cast()` method will very likely produce null values as a result, because it cannot parse dates that are outside the ISO-8601 format. If that is your case, then you should use the `to_date()` function, which allows you to specify the exact format of your dates.

There are many examples of date formats which are outside of the ISO-8601 format. Like:

- In Brazil and Spain, dates are formatted as “Day/Month/Year”. Example: “23/04/2022” for April 23, 2022.
- In Japan, dates are formatted as “year month day (weekday)”, with the Japanese characters meaning “year”, “month” and “day” inserted after the numerals. Example: 2008□12□31□ (□) for “Wednesday 31 December 2008”.
- Many websites display dates using the full name of the month, like “November 18, 2023”. This is an important fact considering that web-scraping is a real and important area of data analysis these days.

I will describe at Section 11.3 how you can use the `to_date()` function to convert dates that are outside of the ISO format to `DateTime` values. But for now, for simplicity sake, I will consider only strings that contains date in the ISO format.

As a first example, lets consider the `DataFrame` `df` below:

```

from pyspark.sql import Row

data = [
    Row(date_registered = "2021-01-01"),
    Row(date_registered = "2021-01-01"),
    Row(date_registered = "2021-01-02"),
    Row(date_registered = "2021-01-03")
]

df = spark.createDataFrame(data)

```

If we look at the DataFrame schema of df, we can see that the date\_registered column is currently being interpreted as a column of type StringType:

```
df.printSchema()
```

```

root
 |-- date_registered: string (nullable = true)

```

Since the dates from the date\_registered column are formatted like the ISO-8601 standard, we can safely use cast() to get a column of type DateType. And if we look again at the DataFrame schema after the transformation, we can certify that the date\_registered column is in fact now, a column of type DateType.

```

from pyspark.sql.functions import col

df = df.withColumn(
    'date_registered',
    col('date_registered').cast('date')
)

df.show()

```

```

+-----+
|date_registered|
+-----+
|    2021-01-01 |
|    2021-01-01 |
|    2021-01-02 |
|    2021-01-03 |
+-----+

```

```
df.printSchema()
```

```
root
|-- date_registered: date (nullable = true)
```

### 11.1.2 From datetime values

A datetime value is a value that contains both a date component and a time component. But you can obviously extract just the date component from a datetime value. Let's use the following DataFrame as example:

```
from pyspark.sql import Row
from datetime import datetime

data = [
    {'as_datetime': datetime(2021, 6, 12, 10, 0, 0)},
    {'as_datetime': datetime(2021, 6, 12, 18, 0, 0)},
    {'as_datetime': datetime(2021, 6, 13, 7, 0, 0)},
    {'as_datetime': datetime(2021, 6, 14, 19, 30, 0)}
]

df = spark.createDataFrame(data)
df.printSchema()
```

```
root
|-- as_datetime: timestamp (nullable = true)
```

You can extract the date component from the `as_datetime` column by directly casting the column into the `DateType` type. Like you would normally do with a string column.

```
df.withColumn('date_component', col('as_datetime').cast('date'))\
    .show()
```

```
+-----+-----+
|      as_datetime|date_component|
+-----+-----+
|2021-06-12 10:00:00|    2021-06-12|
|2021-06-12 18:00:00|    2021-06-12|
|2021-06-13 07:00:00|    2021-06-13|
```

```
|2021-06-14 19:30:00|    2021-06-14|
+-----+-----+
```

### 11.1.3 From individual components

If you have 3 columns in your DataFrame, one for each component of a date value (day, month, year), you can group these components together to form date values. In pyspark you do this by using the `make_date()` function.

To use this function, you just list the columns of each component in the following order: year, month and day. Like in this example:

```
from pyspark.sql.functions import make_date
df3 = spark.createDataFrame([
    Row(day=14,month=2,year=2021),
    Row(day=30,month=4,year=2021),
    Row(day=2,month=5,year=2021),
    Row(day=6,month=5,year=2021)
])

df3\
    .withColumn(
        'as_date',
        make_date(
            col('year'),
            col('month'),
            col('day')
        )
    )\
    .show()
```

```
+---+-----+-----+
|day|month|year|  as_date|
+---+-----+-----+
| 14|    2|2021|2021-02-14|
| 30|    4|2021|2021-04-30|
|  2|    5|2021|2021-05-02|
|  6|    5|2021|2021-05-06|
+---+-----+-----+
```

## 11.2 Creating datetime values

Datetime values are values that contains both a date and a time components. These datetime values might also come with a time zone component, although this component is not mandatory.

Different programming languages and frameworks might have different names for these kind of values. Because of that, you might know datetime values by a different name. For example, “timestamps” is also a very popular name for this kind of values. Anyway, in pyspark, datetime (or timestamp) values are interpreted by the `TimestampType` data type.

There are three common ways to create datetime objects, which are:

1. from strings (like "2023-02-05 12:30:00").
2. from integer values (i.e. `IntegerType` and `LongType`).
3. by combining the individual components of a datetime value (day, month, year, hour, minute, second, etc.) to form a complete datetime value.

### 11.2.1 From strings

Again, if your datetime values are being currently stored inside strings, and, you want to convert them to datetime values, you can use the automatic conversion of Spark with the `cast()` column method. However, as we stated at Section 11.1, when you use this path, Spark will always assume that your datetime values follow the ISO-8601 format.

The ISO-8601 standard states that datetime values should always follow the format “YYYY-MM-DD HH:MM:SS Z” (or “Year-Month-Day Hour:Minute:Second TimeZone”). You can see at Figure 11.2, that are components that are mandatory (meaning that they always appear in a datetime value), and components that are optional (meaning that they might appear or might not in a datetime value).

The time zone component is always optional, and because of that, they usually are not present in the datetime values you have. There are some variations and extra characters that might appear at some points, for example, the character ‘T’ might appear to clearly separate the date from the time component, and the character ‘Z’ to separate the time zone from the time component. Also, the time component might have a microseconds section to identify a more precise point in time.

But despite all these variations, datetime values that are ISO-8601 compatible are basically always following this same pattern (or this same order of components) of “Year-Month-Day Hour:Minute:Second TimeZone”. Examples of values that follows this standard are “2014-10-18T16:30:00Z”, “2019-05-09 08:20:05.324” and “2020-01-01 12:30:00 -03”.

This means that if your datetime values are not in this format, if they do not follow the ISO-8601 standard, then, you should not use the `cast()` method. If that is your case, you should use datetime patterns with the `to_timestamp()` function. We describe these assets in more depth at Section 11.3.

## ISO-8601 format for datetimes/timestamps

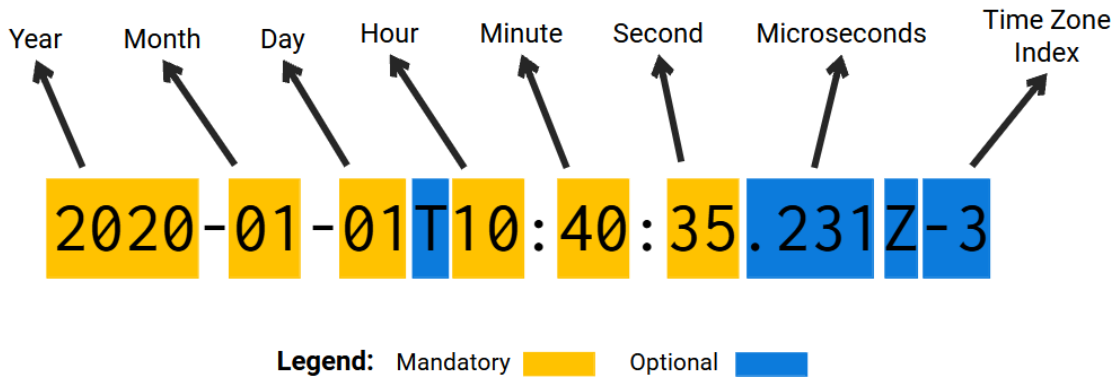


Figure 11.2: The ISO-8601 format for datetime/timestamp values

For now, let's focus on examples that use the ISO-8601 format. As an example, let's use the df2 DataFrame below:

```
data = [  
  Row(datetime_as_string = "2021-02-23T04:41:57Z"),  
  Row(datetime_as_string = "2021-05-18T12:30:05Z"),  
  Row(datetime_as_string = "2021-11-13T16:30:00Z"),  
  Row(datetime_as_string = "2021-08-09T00:30:16Z")  
]  
  
df2 = spark.createDataFrame(data)  
df2.printSchema()
```

```
root  
|-- datetime_as_string: string (nullable = true)
```

You can see above, that the datetime\_as\_string column is currently being interpreted as a column of strings. But since the values are in ISO-8601 format, I can use the cast method to directly convert them into timestamp values.

```
df2 = df2\  
  .withColumn(  
    'datetime_values',
```



```

        col('datetime_as_string').cast('timestamp')
    )

df2.printSchema()

```

```

root
 |-- datetime_as_string: string (nullable = true)
 |-- datetime_values: timestamp (nullable = true)

```

```
df2.show()
```

```

+-----+-----+
| datetime_as_string| datetime_values|
+-----+-----+
| 2021-02-23T04:41:57Z| 2021-02-23 01:41:57|
| 2021-05-18T12:30:05Z| 2021-05-18 09:30:05|
| 2021-11-13T16:30:00Z| 2021-11-13 13:30:00|
| 2021-08-09T00:30:16Z| 2021-08-08 21:30:16|
+-----+-----+

```

## 11.2.2 From integers

You can also convert integers directly to datetime values by using the `cast()` method. In this situation, the integers are interpreted as being the number of seconds since the UNIX time epoch, which is midnight of 1 January of 1970 ("1970-01-01 00:00:00"). In other words, the integer 60 will be converted the point in time that is 60 seconds after "1970-01-01 00:00:00", which would be "1970-01-01 00:01:00".

In the example below, the number 1,000,421,325 is converted into 19:48:45 of 13 September of 2001. Because this exact point in time is 1.000421 billion of seconds ahead of the UNIX epoch.

```

df3 = spark.createDataFrame([
    Row(datetime_as_integer = 1000421325),
    Row(datetime_as_integer = 1000423628),
    Row(datetime_as_integer = 500),
    Row(datetime_as_integer = 1000493412)
])

df3 = df3\
    .withColumn(

```

```

        'datetime_values',
        col('datetime_as_integer').cast('timestamp')
    )

df3.show()

```

```

+-----+-----+
|datetime_as_integer|  datetime_values|
+-----+-----+
|          1000421325|2001-09-13 19:48:45|
|          1000423628|2001-09-13 20:27:08|
|              500|1969-12-31 21:08:20|
|          1000493412|2001-09-14 15:50:12|
+-----+-----+

```

However, you probably notice in the example above, that something is odd. Because the number 500 was converted into "1969-12-31 21:08:20", which is in theory, behind the UNIX epoch, which is 1 January of 1970. Why did that happen? The answer is that **your time zone is always taken into account** during a conversion from integers to datetime values!

In the example above, Spark is running on an operating system that is using the America/Sao\_Paulo time zone (which is 3 hours late from international time zone - UTC-3) as the “default time zone” of the system. As a result, integers will be interpreted as being the number of seconds since the UNIX time epoch **minus 3 hours**, which is "1969-12-31 21:00:00". So, in this context, the integer 60 would be converted into "1969-12-31 21:01:00" (instead of the usual "1970-01-01 00:01:00" that you would expect).

That is why the number 500 was converted into "1969-12-31 21:08:20". Because it is 500 seconds ahead of "1969-12-31 21:00:00", which is 3 hours behind the UNIX time epoch.

But what if you wanted to convert your integers into a UTC-0 time zone? Well, you could just set the time zone of your Spark Session to the international time zone before you do the conversion, with the command below:

```

spark.conf.set("spark.sql.session.timeZone", "UTC")

```

But you could also do the conversion anyway and adjust the values later. That is, you just perform the conversion with the `cast()` method, even if the result would be in your current time zone. After that, you add the amount of time necessary to transpose your datetime values to the international time zone (UTC-0).

So in the above example, since I was using the Brasília time zone (UTC-3) during the above conversion, I just need to add 3 hours to all datetime values to get their equivalents values in international time zone. You can do that by using interval expressions, which will be discussed in more depth at [Section 11.5](#).

```

from pyspark.sql.functions import expr
df3 = df3\
    .withColumn(
        'datetime_values_utc0',
        expr("datetime_values + INTERVAL 3 HOURS")
    )

df3.show()

```

```

+-----+-----+-----+
|datetime_as_integer|  datetime_values|datetime_values_utc0|
+-----+-----+-----+
|          1000421325|2001-09-13 19:48:45| 2001-09-13 22:48:45|
|          1000423628|2001-09-13 20:27:08| 2001-09-13 23:27:08|
|              500|1969-12-31 21:08:20| 1970-01-01 00:08:20|
|          1000493412|2001-09-14 15:50:12| 2001-09-14 18:50:12|
+-----+-----+-----+

```

### 11.2.3 From individual components

If you have 6 columns in your DataFrame, one for each component of a datetime value (day, month, year, hour, minute, second), you can group these components together to compose datetime values. We do this in pyspark by using the `make_timestamp()` function.

To use this function, you just list the columns of each component in the following order: year, month, day, hours, minutes, seconds. Like in this example:

```

from pyspark.sql.functions import make_timestamp
df3 = spark.createDataFrame([
    Row(day=14,month=2,year=2021,hour=12,mins=45,secs=0),
    Row(day=30,month=4,year=2021,hour=8,mins=10,secs=0),
    Row(day=2,month=5,year=2021,hour=5,mins=9,secs=12),
    Row(day=6,month=5,year=2021,hour=0,mins=34,secs=4)
])

df3\
    .withColumn(
        'as_datetime',
        make_timestamp(
            col('year'),
            col('month'),

```

```

        col('day'),
        col('hour'),
        col('mins'),
        col('secs')
    )
)\
.show()

```

```

+---+-----+-----+-----+-----+-----+-----+-----+
|day|month|year|hour|mins|secs|          as_datetime|
+---+-----+-----+-----+-----+-----+-----+
| 14|    2|2021|   12|   45|    0|2021-02-14 12:45:00|
| 30|    4|2021|    8|   10|    0|2021-04-30 08:10:00|
|  2|    5|2021|    5|    9|   12|2021-05-02 05:09:12|
|  6|    5|2021|    0|   34|    4|2021-05-06 00:34:04|
+---+-----+-----+-----+-----+-----+-----+

```

## 11.3 Introducing datetime patterns

Every time you have strings that contains dates and datetime values that are outside of the ISO-8601 format, you usually have to use datetime patterns to convert these strings to date or datetime values. In other words, in this section I will describe how you can use datetime patterns to convert string values (that are outside of the ISO-8601 format) into dates or datetimes values.

Despite the existence of an international standard (like ISO-8601), the used date and datetime formats varies accross different countries around the world. There are tons of examples of these different formats. For example, in Brazil, dates are usually formatted like “Day/Month/Year”. Not only the order of the date components (day, month and year) are different, but also, the separator character (“/” instead of “-”).

In essence, a datetime pattern is a string pattern that describes a specific date or datetime format. This means that we can use a datetime pattern to describe any date or datetime format. A datetime pattern is a string value that is constructed by grouping letters together. Each individual letter represents an individual component of a date or a datetime value.

You can see at Table 11.1 a list of the most common used letters in datetime patterns. If you want, you can also see the full list of possible letters in datetime patterns by visiting the Spark Datetime Patterns page<sup>1</sup>.

---

<sup>1</sup><https://spark.apache.org/docs/latest/sql-ref-datetime-pattern.html>

Table 11.1: List of letters to represent date and datetime components

Letter	Meaning	Example of a valid value
G	era	AD; Anno Domini
y	year	2020; 20
D	day-of-year	189
M/L	month-of-year	7; 07; Jul; July
d	day-of-month	28
Q/q	quarter-of-year	3; 03; Q3; 3rd quarter
E	day-of-week	Tue; Tuesday
F	aligned day of week	3
a	am-pm-of-day	PM
h	clock-hour-of-am-pm	12
K	hour-of-am-pm	0
k	clock-hour-of-day	0
H	hour-of-day	0
m	minute-of-hour	30
s	second-of-minute	55
S	fraction-of-second	978
V	time-zone ID	America/Los_Angeles; Z; -08:30
z	time-zone name	Pacific Standard Time; PST
O	localized zone-offset	GMT+8; GMT+08:00; UTC-08:00;
X	zone-offset 'Z' for zero	Z; -08; -0830; -08:30; -083015; -08:30:15;
x	zone-offset	+0000; -08; -0830; -08:30; -083015; -08:30:15;
Z	zone-offset	+0000; -0800; -08:00;

### 11.3.1 Using datetime patterns to get date values

Following Table 11.1, if we had a date in the format “Day, Month of Year”, like “12, November of 1997”, we would use the letters d, M and y for each of the three components that are present in this format. In fact, let’s create a DataFrame with this exact value, and let’s demonstrate how could you convert it to a DateType value.

```
data = [ {"date": "12, November of 1997"} ]
weird_date = spark.createDataFrame(data)
weird_date.show()
```

```
+-----+
|               date|
+-----+
```

```
|12, November of 1997|
+-----+
```

To convert it from the StringType to DateType we have to use the `to_date()` Spark SQL function. Because with this function, we can provide the datetime pattern that describes the exact format that these dates use. But before we use `to_date()`, we need to build a datetime pattern to use.

The date example above (“12, November of 1997”) starts with a two-digit day number. That is why we begin our datetime pattern with two d’s, to represent this section of the date. After that we have a comma and a space followed by the month name. However, both month name and the year number at the end of the date are in their full formats, instead of their abbreviated formats.

Because of that, we need to tell Spark to use the full format instead of the abbreviated format on both of these two components. To do that, we use four M’s and four y’s, instead of just two. At last, we have a literal “of” between the month name and the year name, and to describe this specific section of the date, we insert 'of' between the M’s and y’s.

In essence, we have the datetime pattern "dd, MMMM 'of' yyyy". Now, we can just use `to_date()` with this datetime pattern to convert the string value to a date value.

```
from pyspark.sql.functions import to_date
pattern = "dd, MMMM 'of' yyyy"
weird_date = weird_date\
    .withColumn("date", to_date(col("date"), pattern))

weird_date.show()
```

```
+-----+
|      date|
+-----+
|1997-11-12|
+-----+
```

```
weird_date.printSchema()
```

```
root
|-- date: date (nullable = true)
```

So, if you have a constant (or fixed) text that is always present in all of your date values, like the “of” in the above example, you must encapsulate this text between quotation marks in your datetime patterns. Also, depending if your components are in a abbreviated format (like “02” for year 2002, or “Jan” for

the January month), or in a full format (like the year “1997” or the month “October”), you might want to repeat the letters one or two times (to use abbreviated formats), or you might want to repeat it four times (to use the full formats). In other words, if you use less than 4 pattern letters, then, Spark will use the short text form, typically an abbreviation form of the component you are referring to *Apache Spark Official Documentation* (2022).

As another example of unusual date formats, let's use the `user_events` DataFrame. You can easily get the data of this DataFrame by downloading the JSON file from the official repository of this book<sup>2</sup>. After you downloaded the JSON file that contains the data, you can import it to your Spark session with the commands below:

```
from pyspark.sql.types import StructType, StructField
from pyspark.sql.types import DoubleType, StringType
from pyspark.sql.types import LongType, TimestampType, DateType
path = "../Data/user-events.json"
schema = StructType([
    StructField('dateOfEvent', StringType(), False),
    StructField('timeOfEvent', StringType(), False),
    StructField('userId', StringType(), False),
    StructField('nameOfEvent', StringType(), False)
])

user_events = spark.read\
    .json(path, schema = schema)

user_events.show()
```

```
+-----+-----+-----+-----+
|dateOfEvent|      timeOfEvent|      userId|      nameOfEvent|
+-----+-----+-----+-----+
| 15/06/2022|15/06/2022 14:33:10|b902e51e-d043-4a6...|      entry|
| 15/06/2022|15/06/2022 14:40:08|b902e51e-d043-4a6...|      click: shop|
| 15/06/2022|15/06/2022 15:48:41|b902e51e-d043-4a6...|select: payment-m...|
+-----+-----+-----+-----+
```

The `dateOfEvent` column of this DataFrame contains date values in the Brazilian format “Day/Month/Year”. To describe this date format, we can use the datetime pattern “dd/MM/yyyy”.

```
date_pattern = "dd/MM/yyyy"
user_events = user_events\
    .withColumn('dateOfEvent', to_date(col('dateOfEvent'), date_pattern))
```

<sup>2</sup><https://github.com/pedropark99/Introd-pyspark/tree/main/Data>

```
user_events.show()
```

```
+-----+-----+-----+-----+
|dateOfEvent|    timeOfEvent|    userId|    nameOfEvent|
+-----+-----+-----+-----+
| 2022-06-15|15/06/2022 14:33:10|b902e51e-d043-4a6...|    entry|
| 2022-06-15|15/06/2022 14:40:08|b902e51e-d043-4a6...|    click: shop|
| 2022-06-15|15/06/2022 15:48:41|b902e51e-d043-4a6...|select: payment-m...|
+-----+-----+-----+-----+
```

### 11.3.2 Using datetime patterns to get datetime/timestamp values

Furthermore, the `user_events` table also contains the `timeOfEvent` column, which is a column of datetime (or timestamp) values, also in the usual Brazilian format “Day/Month/Year Hour:Minutes:Seconds”. Following Table 11.1, we can describe this datetime format with the pattern “dd/MM/yyyy HH:mm:ss”.

When you have a column of string values that you want to convert into the `TimestampType` type, but your values are not in ISO-8601 format, you should use the `to_timestamp()` function (which also comes from the `pyspark.sql.functions` module) together with a datetime pattern to describe the actual format of your values.

You use the `to_timestamp()` function in the same way you would use the `to_date()` function, you reference the name of the column you want to convert into a column of timestamp values, and you also provide a datetime pattern to be used in the conversion. The difference between the two functions is solely on the type of columns they produce. The `to_timestamp()` function always produce a column of type `TimestampType` as a result, while the `to_date()` function produces a column of type `DateType`.

```
from pyspark.sql.functions import to_timestamp
datetime_pattern = "dd/MM/yyyy HH:mm:ss"
user_events = user_events\
    .withColumn(
        'timeOfEvent',
        to_timestamp(col('timeOfEvent'), datetime_pattern)
    )

user_events.show()
```

```
+-----+-----+-----+-----+
|dateOfEvent|    timeOfEvent|    userId|    nameOfEvent|
+-----+-----+-----+-----+
```



```
| 2022-06-15|2022-06-15 14:33:10|b902e51e-d043-4a6...|entry|
| 2022-06-15|2022-06-15 14:40:08|b902e51e-d043-4a6...|click: shop|
| 2022-06-15|2022-06-15 15:48:41|b902e51e-d043-4a6...|select: payment-m...|
+-----+-----+-----+-----+-----+
```

## 11.4 Extracting date or datetime components

One very common operation when dealing with dates and datetime values is extracting components from these values. For example, you might want to create a new column that contains the day component from all of your dates. Or maybe, the month... anyway.

Thankfully, pyspark made it pretty easy to extract these kinds of components. You just use the corresponding function to each component! All of these functions come from the `pyspark.sql.functions` module. As a quick list, the functions below are used to extract components from both dates and datetime values:

- `year()`: extract the year of a given date or datetime as integer.
- `month()`: extract the month of a given date or datetime as integer.
- `dayofmonth()`: extract the day of the month of a given date or datetime as integer.
- `dayofweek()`: extract the day of the week of a given date or datetime as integer. The integer returned will be inside the range 1 (for a Sunday) through to 7 (for a Saturday).
- `dayofyear()`: extract the day of the year of a given date or datetime as integer.
- `quarter()`: extract the quarter of a given date or datetime as integer.

On the other hand, the list of functions below only applies to datetime values, because they extract time components (which are not present on date values). These functions also come to the `pyspark.sql.functions` module:

- `hour()`: extract the hours of a given datetime as integer.
- `minute()`: extract the minutes of a given datetime as integer.
- `second()`: extract the seconds of a given datetime as integer.

In essence, you just apply these functions at the column that contains your date and datetime values, and you get as output the component you want. As an example, let's go back to the `user_events` DataFrame and extract the day, month and year components of each date, into separate columns:

```
from pyspark.sql.functions import (
    dayofmonth,
    month,
    year
)
```

```

user_events\
  .withColumn('dayOfEvent', dayofmonth(col('dateOfEvent')))\
  .withColumn('monthOfEvent', month(col('dateOfEvent')))\
  .withColumn('yearOfEvent', year(col('dateOfEvent')))\
  .select(
    'dateOfEvent', 'dayOfEvent',
    'monthOfEvent', 'yearOfEvent'
  )\
  .show()

```

```

+-----+-----+-----+-----+
|dateOfEvent|dayOfEvent|monthOfEvent|yearOfEvent|
+-----+-----+-----+-----+
| 2022-06-15|      15|         6|      2022|
| 2022-06-15|      15|         6|      2022|
| 2022-06-15|      15|         6|      2022|
+-----+-----+-----+-----+

```

## 11.5 Adding time to date and datetime values with interval expressions

Another very common operation is to add some amount of time to a date or datetime values. For example, you might want to advance your dates by 3 days. Or, delay your datetime values by 3 hours. In pyspark, you can perform this kind by either using functions or interval expressions.

When we talk about functions available through the `pyspark.sql.functions` module, we have `date_add()` and `date_sub()`, which you can use to add or subtract days to a date, and `add_months()`, which you can use to add months to a date. Yes, the naming pattern of these functions is a little bit weird, but let's ignore this fact.

```

from pyspark.sql.functions import (
    date_add,
    date_sub,
    add_months
)

user_events\
  .withColumn('timePlus10Days', date_add(col('timeOfEvent'), 10))\
  .withColumn('timeMinus5Days', date_sub(col('timeOfEvent'), 5))\
  .withColumn('timePlus8Months', add_months(col('timeOfEvent'), 8))\

```

```
.select(
    'timeOfEvent', 'timePlus10Days',
    'timeMinus5Days', 'timePlus8Months'
)\
.show()
```

```
+-----+-----+-----+-----+
|      timeOfEvent|timePlus10Days|timeMinus5Days|timePlus8Months|
+-----+-----+-----+-----+
|2022-06-15 14:33:10|    2022-06-25|    2022-06-10|    2023-02-15|
|2022-06-15 14:40:08|    2022-06-25|    2022-06-10|    2023-02-15|
|2022-06-15 15:48:41|    2022-06-25|    2022-06-10|    2023-02-15|
+-----+-----+-----+-----+
```

Now, you can use interval expressions to add whatever time unit you want (years, months, days, hours, minutes or seconds) to either a date or datetime value. However, interval expressions are only available at the Spark SQL level. As a consequence, to use an interval expression, you must use `expr()` or `spark.sql()` (that we introduced at Chapter 7) to get access to Spark SQL. An interval expression follows this pattern:

**INTERVAL 3 HOURS**

An interval expression is an expression that begins with the `INTERVAL` keyword, and then, it is followed by a number that specifies an amount. You define how much exactly this amount represents by using another keyword that specifies the unit of time to be used. So the keyword `HOUR` specifies that you want to use the “hour” unit of time.

Having this in mind, the expression `INTERVAL 3 HOURS` defines a 3 hours interval expression. In contrast, the expression `INTERVAL 3 SECONDS` represents 3 seconds, and the expression `INTERVAL 90 MINUTES` represents 90 minutes, and so on.

By having an interval expression, you can just add or subtract this interval to your column to change the timestamp you have.

```
from pyspark.sql.functions import expr
user_events\
    .withColumn(
        'timePlus3Hours',
        expr('timeOfEvent + INTERVAL 3 HOURS')
    )\
    .withColumn(
        'timePlus2Years',
```

```

    expr('timeOfEvent + INTERVAL 2 YEARS')
)\
.select('timeOfEvent', 'timePlus3Hours', 'timePlus2Years')\
.show()

```

```

+-----+-----+-----+
|      timeOfEvent|      timePlus3Hours|      timePlus2Years|
+-----+-----+-----+
|2022-06-15 14:33:10|2022-06-15 17:33:10|2024-06-15 14:33:10|
|2022-06-15 14:40:08|2022-06-15 17:40:08|2024-06-15 14:40:08|
|2022-06-15 15:48:41|2022-06-15 18:48:41|2024-06-15 15:48:41|
+-----+-----+-----+

```

## 11.6 Calculating differences between dates and datetime values

Another very common operation is to calculate how much time is between two dates or two datetimes. In other words, you might want to calculate how many days the date "2023-05-12" is far ahead of "2023-05-05", or how many hours the timestamp "2023-07-10 13:13:00" is ahead of "2023-07-10 05:18:00".

When you have two columns of type `DateType` (or `TimestampType`), you subtract one by the other directly to get the difference between the values of these columns. As an example, let's use the `df4` below:

```

from random import randint, seed
from datetime import (
    date,
    datetime,
    timedelta
)

seed(10)
dates = [
    date(2023,4,1) + timedelta(days = randint(1,39))
    for i in range(4)
]

datetimes = [
    datetime(2023,4,1,8,14,54) + timedelta(hours = randint(1,39))
    for i in range(4)
]

```

```

]

df4 = spark.createDataFrame([
    Row(
        date1 = date(2023,4,1),
        date2 = dates[i],
        datetime1 = datetime(2023,4,1,8,14,54),
        datetime2 = datetimes[i]
    )
    for i in range(4)
])

df4.show()

```

```

+-----+-----+-----+-----+
|   date1|   date2|   datetime1|   datetime2|
+-----+-----+-----+-----+
|2023-04-01|2023-05-08|2023-04-01 08:14:54|2023-04-02 21:14:54|
|2023-04-01|2023-04-04|2023-04-01 08:14:54|2023-04-01 09:14:54|
|2023-04-01|2023-04-29|2023-04-01 08:14:54|2023-04-01 22:14:54|
|2023-04-01|2023-05-02|2023-04-01 08:14:54|2023-04-02 14:14:54|
+-----+-----+-----+-----+

```

If we subtract columns date2 by date1 and datetime2 by datetime1, we get two new columns of type `DayTimeIntervalType`. These new columns represent the interval of time between the values of these columns.

```

df4\
  .select('date1', 'date2')\
  .withColumn('datediff', col('date2') - col('date1'))\
  .show()

```

```

+-----+-----+-----+
|   date1|   date2|   datediff|
+-----+-----+-----+
|2023-04-01|2023-05-08|INTERVAL '37' DAY|
|2023-04-01|2023-04-04|INTERVAL '3' DAY|
|2023-04-01|2023-04-29|INTERVAL '28' DAY|
|2023-04-01|2023-05-02|INTERVAL '31' DAY|
+-----+-----+-----+

```

```
df4\
  .select('datetime1', 'datetime2')\
  .withColumn('datetimediff', col('datetime2') - col('datetime1'))\
  .show()
```

```
+-----+-----+-----+
|      datetime1|      datetime2|      datetimediff|
+-----+-----+-----+
|2023-04-01 08:14:54|2023-04-02 21:14:54|INTERVAL '1 13:00...|
|2023-04-01 08:14:54|2023-04-01 09:14:54|INTERVAL '0 01:00...|
|2023-04-01 08:14:54|2023-04-01 22:14:54|INTERVAL '0 14:00...|
|2023-04-01 08:14:54|2023-04-02 14:14:54|INTERVAL '1 06:00...|
+-----+-----+-----+
```

But another very common way to calculate this difference is to convert the datetime values to seconds. Then, you subtract the calculated seconds, so you get as output, the difference in seconds between the two datetime values.

To use this method, you first use the `unix_timestamp()` function to convert the timestamps into seconds as integer values, then, you subtract these new integer values that were created.

```
from pyspark.sql.functions import unix_timestamp
df4 = df4\
  .select('datetime1', 'datetime2')\
  .withColumn('datetime1', unix_timestamp(col('datetime1')))\
  .withColumn('datetime2', unix_timestamp(col('datetime2')))\
  .withColumn('diffinseconds', col('datetime2') - col('datetime1'))

df4.show()
```

```
+-----+-----+-----+
| datetime1| datetime2|diffinseconds|
+-----+-----+-----+
|1680347694|1680480894|      133200|
|1680347694|1680351294|       3600|
|1680347694|1680398094|      50400|
|1680347694|1680455694|     108000|
+-----+-----+-----+
```

Now that we have the `diffinseconds` column, which represents the difference **in seconds** between `datetime1` and `datetime2` columns, we can divide this `diffinseconds` column by 60 to get the difference in minutes, or divide by 3600 to get the difference in hours, etc.

```
df4\
    .withColumn('diffinminutes', col('diffinseconds') / 60)\
    .withColumn('diffinhours', col('diffinseconds') / 3600)\
    .show()
```

```
+-----+-----+-----+-----+-----+
| datetime1| datetime2|diffinseconds|diffinminutes|diffinhours|
+-----+-----+-----+-----+-----+
|1680347694|1680480894|      133200|        2220.0|         37.0|
|1680347694|1680351294|       3600|         60.0|          1.0|
|1680347694|1680398094|      50400|        840.0|         14.0|
|1680347694|1680455694|     108000|       1800.0|         30.0|
+-----+-----+-----+-----+-----+
```

## 11.7 Getting the now and today values

If for some reason, you need to know which timestamp is now, or which date is today, you can use the `current_timestamp()` and `current_date()` functions from `pyspark.sql.functions` module. When you execute these functions, they output the current timestamp and date in your system.

```
from pyspark.sql.types import StructType, StructField, StringType
from pyspark.sql.functions import (
    current_date,
    current_timestamp,
    lit
)

data = [Row(today="", now="")]
schema = StructType([
    StructField('today', StringType(), True),
    StructField('now', StringType(), True)
])

spark.createDataFrame(data, schema = schema)\
    .withColumn('today', lit(current_date()))\
    .withColumn('now', lit(current_timestamp()))\
    .show()
```

```
+-----+-----+
```

today	now
+-----+-----+	
2023-12-19 2023-12-19 21:31:...	
+-----+-----+	



## 12 Introducing window functions

Spark offers a set of tools known as *window functions*. These tools are essential for an extensive range of tasks, and you should know them. But what are they?

Window functions in Spark are a set functions that performs calculations over windows of rows from your DataFrame. This is not a concept exclusive to Spark. In fact, window functions in Spark are essentially the same as [window functions in MySQL](#)<sup>1</sup>.

When you use a window function, the rows of your DataFrame are divided into multiple windows. Each window contains a specific range of rows from the DataFrame. In this context, a window function is a function that receives a window (or a range of rows) as input, and calculates an aggregate or a specific index based on the set of rows that is contained in this input window.

You might find this description very similar to what `groupby()` and `agg()` methods do when combined together. And yes... To some extent, the idea of windows in a DataFrame is similar (but not identical) to the idea of “groups” created by *group by* functions, such as the `DataFrame.groupby()` method from `pyspark` (that we presented at Section 5.10.4), or the `DataFrame.groupby()`<sup>2</sup> method from `pandas`, and also, to `dplyr::group_by()`<sup>3</sup> from the `tidyverse` framework. You will see further in this chapter how window functions differ from these operations.

### 12.1 How to define windows

In order to use a window function you need to define the windows of your DataFrame first. You do this by creating a Window object in your session.

Every window object have two components, which are partitioning and ordering, and you specify each of these components by using the `partitionBy()` and `orderBy()` methods from the Window class. In order to create a Window object, you need to import the Window class from the `pyspark.sql.window` module:

```
from pyspark.sql.window import Window
```

---

<sup>1</sup><https://dev.mysql.com/doc/refman/8.0/en/window-functions-usage.html>

<sup>2</sup><https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.groupby.html>

<sup>3</sup>[https://dplyr.tidyverse.org/reference/group\\_by.html](https://dplyr.tidyverse.org/reference/group_by.html)

Over the next examples, I will be using the `transf` DataFrame that we presented at Chapter 5. If you don't remember how to import/get this DataFrame into your session, come back to Section 5.4.

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

```
+-----+-----+-----+-----+-----+
|dateTransfer|  datetimeTransfer|clientNumber|transferValue|transferCurrency|
+-----+-----+-----+-----+-----+
|  2022-12-31|2022-12-31 14:00:24|      5516|      7794.31|          zing f|
|  2022-12-31|2022-12-31 10:32:07|      4965|       7919.0|          zing f|
|  2022-12-31|2022-12-31 07:37:02|      4608|       5603.0|        dollar $|
|  2022-12-31|2022-12-31 07:35:05|      1121|       4365.22|        dollar $|
|  2022-12-31|2022-12-31 02:53:44|      1121|       4620.0|        dollar $|
+-----+-----+-----+-----+-----+
```

only showing top 5 rows

... with 5 more columns: `transferID`, `transferLog`, `destinationBankNumber`  
`destinationBankBranch`, `destinationBankAccount`

Now, let's create a window object using the `transf` DataFrame as our target. This DataFrame describes a set of transfers made in a fictitious bank. So a reasonable way of splitting this DataFrame is by day. That means that we can split this DataFrame into groups (or ranges) of rows by using the `dateTransfer` column. As a result, each partition in the `dateTransfer` column will create/identify a different window in this DataFrame.

```
window_spec = Window.partitionBy('dateTransfer')
```

The above window object specifies that each unique value present in the `dateTransfer` column identifies a different window frame in the `transf` DataFrame. Figure 12.1 presents this idea visually. So each partition in the `dateTransfer` column creates a different window frame. And each window frame will become an input to a window function (when we use one).

Until this point, defining windows are very much like defining groups in your DataFrame with *group by* functions (i.e. windows are very similar to groups). But in the above example, we specified only the partition component of the windows. The partitioning component of the window object specifies which partitions of the DataFrame are translated into windows. In the other hand, the ordering component of the window object specifies how the rows within the window are ordered.

Defining the ordering component becomes very important when we are working with window functions that outputs (or that uses) indexes. As an example, you might want to use in your calculations the first (or the *nth*) row in each window. In a situation like this, the order in which these rows are founded inside the window affects directly the output of your window function. That is why the ordering component matters.

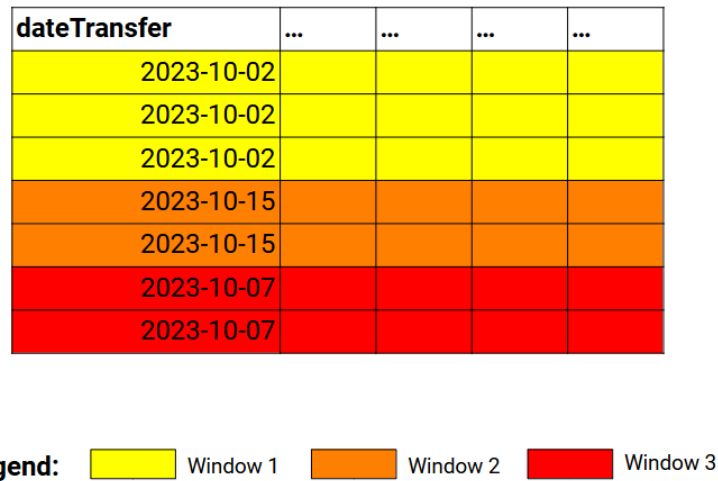


Figure 12.1: Visualizing the window frames - Part 1

For example, we can say that the rows within each window should be in descending order according to the datetimeTransfer column:

```
from pyspark.sql.functions import col
window_spec = Window\
    .partitionBy('dateTransfer')\
    .orderBy(col('datetimeTransfer')).desc())
```

With the above snippet, we are not only specifying how the window frames in the DataFrame are created (with the `partitionBy()`), but we are also specifying how the rows within the window are sorted (with the `orderBy()`). If we update our representation with the above window specification, we get something similar to Figure 12.2:

Is worth mentioning that, both `partitionBy()` and `orderBy()` methods accepts multiple columns as input. In other words, you can use a combination of columns both to define how the windows in your DataFrame will be created, and how the rows within these windows will be sorted.

As an example, the window specification below is saying: 1) that a window frame is created for each unique combination of `dateTransfer` and `clientNumber`; 2) that the rows within each window are ordered accordingly to `transferCurrency` (ascending order) and `datetimeTransfer` (descending order).

dateTransfer	datetimeTransfer	...	...	...
2023-10-02	2023-10-02 23:59:58			
2023-10-02	2023-10-02 20:12:43			
2023-10-02	2023-10-02 19:02:12			
2023-10-15	2023-10-15 21:42:32			
2023-10-15	2023-10-15 15:04:02			
2023-10-07	2023-10-07 23:45:21			
2023-10-07	2023-10-07 21:21:08			

**Legend:**  Window 1  Window 2  Window 3

Figure 12.2: Visualizing the window frames - Part 2

```

window_spec = Window\
    .partitionBy('dateTransfer', 'clientNumber')\
    .orderBy(
        col('transferCurrency').asc(),
        col('datetimeTransfer').desc()
    )

```

### 12.1.1 Partitioning or ordering or none

Is worth mentioning that both partitioning and ordering components of the window specification **are optional**. You can create a window object that contains only a partitioning component defined, or, only a ordering component, or, in fact, a window object that basically have neither of them defined.

As an example, all three objects below (w1, w2 and w3) are valid window objects. w1 have only the partition component defined, while w2 have only the ordering component defined. However, w3 have basically none of them defined, because w3 is partitioned by nothing. In a situation like this, a single window is created, and this window covers the entire DataFrame. It covers all the rows at once. Is like you were not using any window at all.

```
w1 = Window.partitionBy('x')
w2 = Window.orderBy('x')
w3 = Window.partitionBy()
```

So just be aware of this. Be aware that you can cover the entire DataFrame into a single window. Be aware that if you use a window object with neither components defined (`Window.partitionBy()`) your window function basically works with the entire DataFrame at once. In essence, this window function becomes similar to a normal aggregating function.

## 12.2 Introducing the `over()` clause

In order to use a window function you **need to combine an over clause with a window object**. If you pair these two components together, then, the function you are using becomes a window function.

Since we know now how to define window objects for our DataFrame, we can actually create and use this object to access window functionality, by pairing this window object with an `over()` clause.

In pyspark this `over()` clause is actually a method from the Column class. Since all aggregating functions available from the `pyspark.sql.functions` module produces a new Column object as output, we tend to use the `over()` method right after the function call.

For example, if we wanted to calculate the mean of `x` with the `mean()` function, and we had a window object called `window_spec`, we could use the `mean()` as a window function by writing `mean(col('x')).over(window_spec)`.

```
from pyspark.sql.window import Window
from pyspark.sql.functions import mean, col
window_spec = Window\
    .partitionBy('y', 'z')\
    .orderBy('t')

mean(col('x')).over(window_spec)
```

If you see this `over()` method after a call of an aggregating function (such as `sum()`, `mean()`, etc.), then, you know that this aggregating function is being called as a window function.

The `over()` clause is also available in Spark SQL as the SQL keyword `OVER`. This means that you can use window functions in Spark SQL as well. But in Spark SQL, you write the window specification inside parentheses after the `OVER` keyword, and you specify each component with `PARTITION BY` AND `ORDER BY` keywords. We could replicate the above example in Spark SQL like this:

```
SELECT mean(x) OVER (PARTITION BY y, z ORDER BY t ASC)
```

## 12.3 Window functions vs *group by* functions

Despite their similarities, window functions and *group by* functions are used for different purposes. One big difference between them, is that when you use `groupby()` + `agg()` you get one output row per each input group of rows, but in contrast, a window function outputs one row per input row. In other words, for a window of  $n$  input rows a window function outputs  $n$  rows that contains the same result (or the same aggregate result).

For example, lets suppose you want to calculate the total value transfered within each day. If you use a `groupby()` + `agg()` strategy, you get as result a new DataFrame containing one row for each unique date present in the `dateTransfer` column:

```
from pyspark.sql.functions import sum
transf\
    .orderBy('dateTransfer')\
    .groupBy('dateTransfer')\
    .agg(sum(col('transferValue')).alias('dayTotalTransferValue'))\
    .show(5)
```

```
+-----+-----+
|dateTransfer|dayTotalTransferValue|
+-----+-----+
| 2022-01-01|          39630.7|
| 2022-01-02|          70031.46|
| 2022-01-03|    50957.869999999995|
| 2022-01-04|          56068.34|
| 2022-01-05|          47082.04|
+-----+-----+
```

only showing top 5 rows

On the other site, if you use `sum()` as a window function instead, you get as result one row for each transfer. That is, you get one row of output for each input row in the `transf` DataFrame. The value that is present in the new column created (`dayTotalTransferValue`) is the total value transfered for the window (or the range of rows) that corresponds to the date in the `dateTransfer` column.

In other words, the value 39630.7 below corresponds to the sum of the `transferValue` column when `dateTransfer == "2022-01-01"`:

```
window_spec = Window.partitionBy('dateTransfer')
transf\
    .select('dateTransfer', 'transferID', 'transferValue')\
    .withColumn(
```

```

    'dayTotalTransferValue',
    sum(col('transferValue')).over(window_spec)
  )\
  .show(5)

```

```

+-----+-----+-----+-----+
|dateTransfer|transferID|transferValue|dayTotalTransferValue|
+-----+-----+-----+-----+
|  2022-01-01|  20221148|      5547.13|          39630.7|
|  2022-01-01|  20221147|      9941.0|          39630.7|
|  2022-01-01|  20221146|      5419.9|          39630.7|
|  2022-01-01|  20221145|      5006.0|          39630.7|
|  2022-01-01|  20221144|      8640.06|          39630.7|
+-----+-----+-----+-----+

```

only showing top 5 rows

You probably already seen this pattern in other data frameworks. As a quick comparison, if you were using the tidyverse framework, you could calculate the exact same result above with the following snippet of R code:

```

transf |>
  group_by(dateTransfer) |>
  mutate(
    dayTotalTransferValue = sum(transferValue)
  )

```

In contrast, you would need the following snippet of Python code to get the same result in the pandas framework:

```

transf['dayTotalTransferValue'] = transf['transferValue']\
  .groupby(transf['dateTransfer'])\
  .transform('sum')

```

## 12.4 Ranking window functions

The functions `row_number()`, `rank()` and `dense_rank()` from the `pyspark.sql.functions` module are ranking functions, in the sense that they seek to rank each row in the input window according to a ranking system. These functions are identical to their [peers in MySQL<sup>4</sup>](https://dev.mysql.com/doc/refman/8.0/en/window-function-descriptions.html#function_row-number) `ROW_NUMBER()`, `RANK()` and `DENSE_RANK()`.

<sup>4</sup>[https://dev.mysql.com/doc/refman/8.0/en/window-function-descriptions.html#function\\_row-number](https://dev.mysql.com/doc/refman/8.0/en/window-function-descriptions.html#function_row-number)

The function `row_number()` simply returns a unique and sequential number to each row in a window, starting from 1. It is a quick way of marking each row with an unique and sequential number.

```
from pyspark.sql.functions import row_number
window_spec = Window\
    .partitionBy('dateTransfer')\
    .orderBy('datetimeTransfer')

transf\
    .select(
        'dateTransfer',
        'datetimeTransfer',
        'transferID'
    )\
    .withColumn('rowID', row_number().over(window_spec))\
    .show(5)
```

```
+-----+-----+-----+-----+
|dateTransfer|  datetimeTransfer|transferID|rowID|
+-----+-----+-----+-----+
|  2022-01-01|2022-01-01 03:56:58|  20221143|   1|
|  2022-01-01|2022-01-01 04:07:44|  20221144|   2|
|  2022-01-01|2022-01-01 09:00:18|  20221145|   3|
|  2022-01-01|2022-01-01 10:17:04|  20221146|   4|
|  2022-01-01|2022-01-01 16:14:30|  20221147|   5|
+-----+-----+-----+-----+
```

only showing top 5 rows

The `row_number()` function is also very useful when you are trying to collect the rows in each window that contains the smallest or biggest value in the window. If the ordering of your window specification is in ascending order, then, the first row in the window will contain the smallest value in the current window. In contrast, if the ordering is in descending order, then, the first row in the window will contain the biggest value in the current window.

This is interesting, because lets suppose you wanted to find the rows that contained the maximum transfer values in each day. A `groupby()` + `agg()` strategy would tell you which are the maximum transfer values in each day. But it would not tell you where are the rows in the DataFrame that contains these maximum values. A Window object + `row_number()` + `filter()` can help you to get this answer.

```
window_spec = Window\
    .partitionBy('dateTransfer')\
    .orderBy(col('transferValue').desc())
```



```
# The row with rowID == 1 is the first row in each window
transf\
  .withColumn('rowID', row_number().over(window_spec))\
  .filter(col('rowID') == 1)\
  .select('dateTransfer', 'rowID', 'transferID', 'transferValue')\
  .show(5)
```

```
+-----+-----+-----+-----+
|dateTransfer|rowID|transferID|transferValue|
+-----+-----+-----+-----+
| 2022-01-01| 1| 20221147| 9941.0|
| 2022-01-02| 1| 20221157| 10855.01|
| 2022-01-03| 1| 20221165| 8705.65|
| 2022-01-04| 1| 20221172| 9051.0|
| 2022-01-05| 1| 20221179| 9606.0|
+-----+-----+-----+-----+
```

only showing top 5 rows

The `rank()` and `dense_rank()` functions are similar to each other. They both rank the rows with integers, just like `row_number()`. But if there is a tie between two rows (that means that both rows have the same value in the ordering column, so it becomes a tie, we do not know which one of these rows should come first), then, these functions will repeat the same number/index for these rows in tie. Lets use the df below as a quick example:

```
data = [
    (1, 3000), (1, 2400),
    (1, 4200), (1, 4200),
    (2, 1500), (2, 2000),
    (2, 3000), (2, 3000),
    (2, 4500), (2, 4600)
]
df = spark.createDataFrame(data, ['id', 'value'])
```

If we apply both `rank()` and `dense_rank()` over this DataFrame with the same window specification, we can see the difference between these functions. In essence, `rank()` leave gaps in the indexes that come right after any tied rows, while `dense_rank()` does not.

```
from pyspark.sql.functions import rank, dense_rank
window_spec = Window\
  .partitionBy('id')\
  .orderBy('value')
```

```
# With rank() there are gaps in the indexes
df.withColumn('with_rank', rank().over(window_spec))\
  .show()
```

```
+---+-----+-----+
| id|value|with_rank|
+---+-----+-----+
| 1| 2400|      1|
| 1| 3000|      2|
| 1| 4200|      3|
| 1| 4200|      3|
| 2| 1500|      1|
| 2| 2000|      2|
| 2| 3000|      3|
| 2| 3000|      3|
| 2| 4500|      5|
| 2| 4600|      6|
+---+-----+-----+
```

```
# With dense_rank() there are no gaps in the indexes
df.withColumn('with_dense_rank', dense_rank().over(window_spec))\
  .show()
```

```
+---+-----+-----+
| id|value|with_dense_rank|
+---+-----+-----+
| 1| 2400|      1|
| 1| 3000|      2|
| 1| 4200|      3|
| 1| 4200|      3|
| 2| 1500|      1|
| 2| 2000|      2|
| 2| 3000|      3|
| 2| 3000|      3|
| 2| 4500|      4|
| 2| 4600|      5|
+---+-----+-----+
```

## 12.5 Agreggating window functions

In essence, all agreggating functions from the `pyspark.sql.functions` module (like `sum()`, `mean()`, `count()`, `max()` and `min()`) can be used as a window function. So you can apply any agreggating function as a window function. You just need to use the `over()` clause with a `Window` object.

We could for example see how much each `transferValue` deviates from the daily mean of transfered value. This might be a valuable information in case you are planning to do some statistical inference over this data. Here is an example of what this would looks like in pyspark:

```
from pyspark.sql.functions import mean
window_spec = Window.partitionBy('dateTransfer')

mean_deviation_expr = (
    col('transferValue')
    - mean(col('transferValue')).over(window_spec)
)

transf\
    .select('dateTransfer', 'transferValue')\
    .withColumn('meanDeviation', mean_deviation_expr)\
    .show(5)
```

```
+-----+-----+-----+
|dateTransfer|transferValue|    meanDeviation|
+-----+-----+-----+
|  2022-01-01|      5547.13|-1057.9866666666658|
|  2022-01-01|       9941.0|  3335.8833333333334|
|  2022-01-01|       5419.9|-1185.2166666666662|
|  2022-01-01|       5006.0|-1599.1166666666659|
|  2022-01-01|      8640.06|  2034.9433333333336|
+-----+-----+-----+
```

only showing top 5 rows

As another example, you might want to calculate how much a specific transfer value represents of represents of the total amount transferred daily. You could just get the total amount transferred daily by applying the `sum()` function over windows partitioned by `dateTransfer`. Then, you just need to divide the current `transferValue` by the result of this `sum()` function, and you get the proportion you are looking for.

```

from pyspark.sql.functions import sum
proportion_expr = (
    col('transferValue')
    / sum(col('transferValue')).over(window_spec)
)

transf\
    .select('dateTransfer', 'transferValue')\
    .withColumn('proportionDailyTotal', proportion_expr)\
    .show(5)

```

```

+-----+-----+-----+
|dateTransfer|transferValue|proportionDailyTotal|
+-----+-----+-----+
| 2022-01-01|      5547.13| 0.1399705278988259|
| 2022-01-01|      9941.0| 0.25084088850310493|
| 2022-01-01|      5419.9| 0.1367601379738435|
| 2022-01-01|      5006.0| 0.1263162144499088|
| 2022-01-01|      8640.06| 0.2180143171833957|
+-----+-----+-----+
only showing top 5 rows

```

## 12.6 Getting the next and previous row with lead() and lag()

There is one pair functions that is worth talking about in this chapter, which are `lead()` and `lag()`. These functions are very useful in the context of windows, because they return the value in the next and previous rows considering your current position in your DataFrame.

These functions basically performs the same operation as their peers `dplyr::lead()` and `dplyr::lag()`<sup>5</sup> from the tidyverse framework. In essence, `lead()` will return the value of the next row, while `lag()` will return the value of the previous row.

```

from pyspark.sql.functions import lag, lead
window_spec = Window\
    .partitionBy('dateTransfer')\
    .orderBy('datetimeTransfer')

lead_expr = lead('transferValue').over(window_spec)
lag_expr = lag('transferValue').over(window_spec)

```

<sup>5</sup><https://dplyr.tidyverse.org/reference/lead-lag.html>

```

transf\
  .withColumn('nextValue', lead_expr)\
  .withColumn('previousValue', lag_expr)\
  .select(
    'datetimeTransfer',
    'transferValue',
    'nextValue',
    'previousValue'
  )\
  .show(5)

```

```

+-----+-----+-----+-----+
| datetimeTransfer|transferValue|nextValue|previousValue|
+-----+-----+-----+-----+
|2022-01-01 03:56:58|      5076.61|  8640.06|          NULL|
|2022-01-01 04:07:44|      8640.06|   5006.0|      5076.61|
|2022-01-01 09:00:18|       5006.0|   5419.9|      8640.06|
|2022-01-01 10:17:04|       5419.9|   9941.0|      5006.0|
|2022-01-01 16:14:30|       9941.0|   5547.13|     5419.9|
+-----+-----+-----+-----+

```

only showing top 5 rows

## References

- Apache Spark Official Documentation*. 2022. Documentation for Apache Spark 3.2.1. <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.
- Damji, Jules, Brooke Wenig, Tathagata Das, and Denny Lee. 2020. *Learning Spark: Lightning-Fast Data Analytics*. Sebastopol, CA: O'Reilly Media.
- Goyvaerts, Jan. 2023. "Regular-Expressions.info." <https://www.regular-expressions.info/>.
- Karau, Holden, Andy Konwinski, Patrick Wendell, and Matei Zaharia. 2015. *Learning Spark: Lightning-Fast Data Analytics*. Sebastopol, CA: O'Reilly Media.