Chapter 6. Working with Different Typesof Data

Data Types:

- Boolean
- Numbers
- Strings
- Dates and timestamps
- Handling null
- Complex types
- User-defined function

Spark is a growing project, and any book (including this one) is a snapshot in time

DataFrame (Dataset) Methods

- a DataFrame is just a Dataset of Row types
- Dataset submodules *DataFrameStatFunctions* and *DataFrameNaFunctions*
    - DataFrameStatFunctions - holds a variety of statistically related functions
    - DataFrameNaFunctions - functions relevant when working with null data

Column Methods - hold a variety of general column-related methods like alias or contains.

org.apache.spark.sql.functions - contains a variety of functions for a range of different data types

lit function -converts a type in another language to its corresponding Spark representation

Booleans

- the foundation for all filtering
- consist of four elements: and, or, true, and false
- build logical statements that evaluate to either true or false
- conditional requirements for when a row of data must either pass the test(evaluate to true) or else it will be filtered out

if Boolean statements are expressed serially (one after the other),Spark will flatten all of these filters into one statement and perform the filter at the same time,creating the and statement for us

```
# in Python
from pyspark.sql.functions import instr
priceFilter=col("UnitPrice")>600descripFilter=instr(df.Description,"POSTAGE")>=
```

```
1df.where(df.StockCode.isin("DOT")).where(priceFilter|descripFilter).show()
```

Gotcha **-** if you're working with null data when creating Boolean expressions

null-safe equivalence test:

```
df.where(col("Description").eqNullSafe("hello")).show()
```

fabricate a contrived example

- let's imagine that we found out that we mis-recorded the quantity in our retail dataset
- true quantity is equal to (the current quantity * the unitprice) + 5.

numerical function as well as the pow function raises a column to the expressed power:

```
# in Python
from pyspark.sql.functions import expr,pow
fabricatedQuantity=pow(col("Quantity")*col("UnitPrice"),2)+5df.select(expr("Cus
tomerId"),fabricatedQuantity.alias("realQuantity")).show(2
```

Another common numerical task is rounding. If you'd like to just round to a whole number,oftentimes you can cast the value to an integer and that will work just fine. However, Spark also has more detailed functions for performing this explicitly and to a certain level of precision. By default, the round function rounds up if you're exactly in between two numbers. You can round down by using the **bround.**

```
# in Python
from pyspark.sql.functions import lit,round,bround
df.select(round(lit("2.5")),bround(lit("2.5"))).show(2
```

Compute the correlation of two columns. For example, we can seethe Pearson correlation coefficient for two columns to see if cheaper things are typically boughtin greater quantities. We can do this through a function as well as through the DataFramestatistic methods

```
from pyspark.sql.functions import corr
df.stat.corr("Quantity", "UnitPrice")
df.select(corr("Quantity", "UnitPrice")).show()
```

calculate either exact or approximate quantiles of your data using the approxQuantile method:

```
# in Python
colName=“UnitPrice"
quantileProbs=
[0.5]relError=0.05df.stat.approxQuantile("UnitPrice",quantileProbs,relError)#
2.51
```

a cross-tabulation

```
df.stat.crosstab("StockCode", "Quantity").show()
```

frequent item pairs

```
df.stat.freqItems(["StockCode", "Quantity"]).show()
```

add a unique ID to each row by using the function **monotonically_increasing_id**. This function generates a unique value for each row, starting with 0:

```
from pyspark.sql.functions import monotonically_increasing_id
df.select(monotonically_increasing_id()).show(2)
```

random data generation tools (e.g., rand(), randn()) with which you can randomly generate data; however, there are potential determinism issues when doing so

Strings

Initcap - function will capitalize every word in a given string when that word is separated from another by a space:

```
from pyspark.sql.functions import initcap
df.select(initcap(col("Description"))).show()
```

adding or removing spaces around a string - you can do this by using *lpad,ltrim, rpad* and *rtrim, trim:*

```
from pyspark.sql.functions import lit, ltrim, rtrim, rpad, lpad, trim
df.select(
    ltrim(lit("    HELLO    ")).alias("ltrim"),
    rtrim(lit("    HELLO    ")).alias("rtrim"),
    trim(lit("    HELLO    ")).alias("trim"),
    lpad(lit("HELLO"), 3, " ").alias("lp"),
    rpad(lit("HELLO"), 10, " ").alias("rp")).show(2)
```

Regular Expressions

two key functions in Spark that you'll need inorder to perform regular expression tasks: regexp_extract and regexp_replace

```
from pyspark.sql.functions import regexp_replace
regex_string = "BLACK|WHITE|RED|GREEN|BLUE"
df.select(
  regexp_replace(col("Description"), regex_string,
"COLOR").alias("color_clean"),
  col("Description")).show(7)
```

Spark provides the translate function to replace these values. This is done at the character level and will replace all instances of a character with the indexed character in the replacement string:

```
from pyspark.sql.functions import translate
df.select(translate(col("Description"), "LEET", "1337"),col("Description"))\
  .show(2)
```

Check for existence. We can do this with the contains method on each column. This will return a Boolean declaring whether the value you specify is in the column's string:

```
from pyspark.sql.functions import instr
containsBlack = instr(col("Description"), "BLACK") >= 1
containsWhite = instr(col("Description"), "WHITE") >= 1
df.withColumn("hasSimpleColor", containsBlack | containsWhite)\
  .where("hasSimpleColor")\
  .select("Description").show(3, False)
```

locate - that returns the integer location (1 based location), then convert that to a Boolean before using it as the same basic feature:

```
from pyspark.sql.functions import expr, locate
simpleColors = ["black", "white", "red", "green", "blue"]
def color_locator(column, color_string):
  return locate(color_string.upper(), column)\
          .cast("boolean")\
          .alias("is_" + color_string)
selectedColumns = [color_locator(df.Description, c) for c in simpleColors]
selectedColumns.append(expr("*")) # has to a be Column type


df.select(*selectedColumns).where(expr("is_white OR is_red"))\
  .select("Description").show(3, False)
```

Time

get the current date and the current timestamps:

```
from pyspark.sql.functions import current_date, current_timestamp
dateDF = spark.range(10)\
  .withColumn("today", current_date())\
  .withColumn("now", current_timestamp())
dateDF.createOrReplaceTempView("dateTable”)
```

Add and subtract five days from today. These functions take a column and then the number of days to either add or subtract as the arguments:

```
from pyspark.sql.functions import date_add, date_sub
dateDF.select(date_sub(col("today"), 5), date_add(col("today"), 5)).show(1)
```

datediff function - will return the number of days in between two dates.
months_between - gives you the number of months between two dates

```
from pyspark.sql.functions import datediff, months_between, to_date
dateDF.withColumn("week_ago", date_sub(col("today"), 7))\
  .select(datediff(col("week_ago"), col("today"))).show(1)


dateDF.select(
    to_date(lit("2016-01-01")).alias("start"),
    to_date(lit("2017-05-22")).alias("end"))\
  .select(months_between(col("start"), col("end"))).show(1)
```

The **to_date** function allowsyou to convert a string to a date, optionally with a specified format. We specify our format in the Java SimpleDateFormat which will be important to reference if you use this function:

```
from pyspark.sql.functions import to_date, lit
spark.range(5).withColumn("date", lit("2017-01-01"))\
  .select(to_date(col("date"))).show(1)
```

Spark will not throw an error if it cannot parse the date; rather, it will just return null. This can be a bit tricky in larger pipelines because you might be expecting your data in one format and getting it in another. To illustrate, let's take a look at the date format that has switched from year-month-day to year-day-month. Spark will fail to parse this date and silently return null instead:

```
dateDF.select(to_date(lit("2016-20-12")),to_date(lit("2017-12-11"))).show(1)
```

Select the first non-null value from a set of columns by using the **coalesce** function. In this case, there are no null values, so it simply returns the first column:

```
from pyspark.sql.functions import coalesce
df.select(coalesce(col("Description"), col("CustomerId"))).show()
```

ifnull allows you to select the second value if the first is null, and defaults to the first. Alternatively, you could use **nullif**, which returns null if the two values are equal or else returns the second if they are not. **nvl** returns the second value if the first is null, but defaults to the first. Finally, **nvl2** returnsthe second value if the first is not null; otherwise, it will return the last specified value

fill function, you can fill one or more columns with a set of values. This can be done by specifying a map—that is a particular value and a set of columns

Replace - replace all values in a certain column according to their current value. The only requirement is that this value be the same type as the original value.

DataFrame with a column complex. We can query it just as we might another DataFrame, the only difference is that we use a *dot syntax* to do so, or the column method **getField**

Split function and specify the delimiter:

```
from pyspark.sql.functions import split
df.select(split(col("Description"), " "), col("Description")).show(2)
```

Manipulate this complex type as another column:

```
df.select(split(col("Description"), " ").alias("array_col"))\
  .selectExpr("array_col[0]").show(2)
```

explode function takes a column that consists of arrays and creates one row (with the rest of the values duplicated) per value in the array:

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


df.withColumn("splitted", split(col("Description"), " "))\
  .withColumn("exploded", explode(col("splitted")))\
  .select("Description", "InvoiceNo", "exploded").show(2)
```

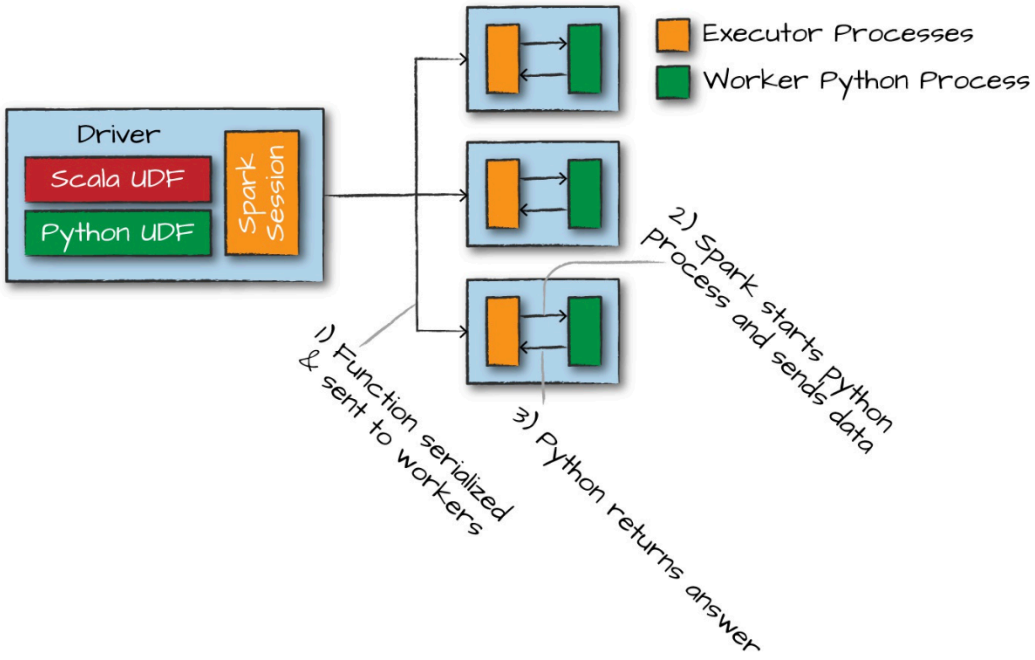
Maps are created by using the map function and key-value pairs of columns. You then can select them just like you might select from an array:

```
from pyspark.sql.functions import create_map
df.select(create_map(col("Description"),
col("InvoiceNo")).alias("complex_map"))\
  .show(2)
```

register a function to make it available as a DataFrame function:

```
from pyspark.sql.types import IntegerType, DoubleType
spark.udf.register("power3py", power3, DoubleType())
```

If you specify the type that doesn't align with the actual type returned by the function, Spark will not throw an error but will just return **null** to designate a failure


Executor Processes
Worker Python Process
Driver
Scala UDF
Python UDF
Spark session
1) Function serialized & sent to workers
2) Spark starts Python process and sends data
3) Python returns answer