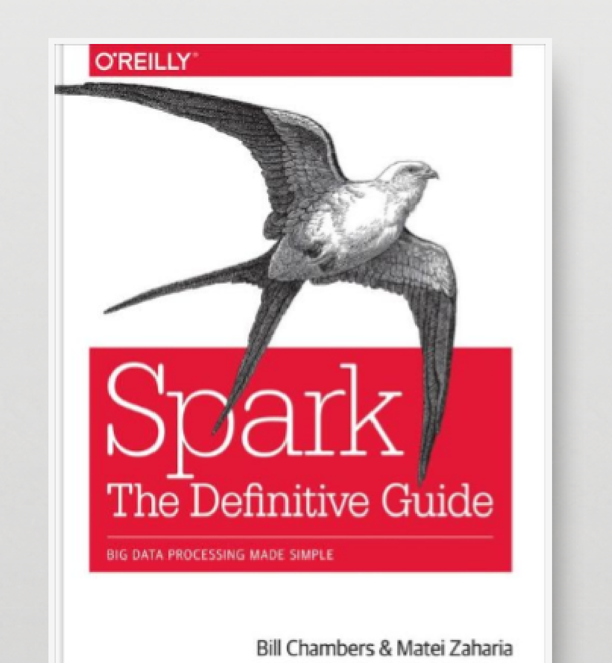
Spark the Definitive Guide 2nd Edition

Chapter 06

Working With Different Types of Data

Basic Structured Operations

Text Book


O'REILLY
Spark
The Definitive Guide
BIG DATA PROCESSING MADE SIMPLE
Bill Chambers & Matei Zaharia

Objectives and Outcomes

- Understand how to build expressions using typed data
- Understand how to use:
  - Booleans
  - Numbers
  - Strings
  - Dates and Timestamps
  - Nulls
  - Complex and user types

Review

- So far:
  - We covered basic DataFrame operations.
  - We learned the simple concepts and tools that you will need to be successful with Spark DataFrames
  - We learned what an expression is
  - We learned the difference between Select and SelectExpr
  - We learned how to add columns and rows to a DataFrame
  - We learned how to take random samples from DataFrames

API Documentation

- Where do all of these functions come from?
  - There is Spark documentation for Datasets in Scala
  - Python API
  - A DataFrame is just a Dataset of type `Row` so you'll end up looking at the Dataset methods
- Column Methods such as `alias` or `contains`
  - Column
  - *org.apache.spark.sql.functions*
- All of these tools exist to transform rows of data in one format or structure to another.

```
df = spark.read.format("csv").option("header","true").optio
df.printSchema()
df.createOrReplaceTempView("dfTable")
```

Schema Output

```
>>> df.printSchema()
root
 |-- InvoiceNo: string (nullable = true)
 |-- StockCode: string (nullable = true)
 |-- Description: string (nullable = true)
 |-- Quantity: integer (nullable = true)
 |-- InvoiceDate: timestamp (nullable = true)
 |-- UnitPrice: double (nullable = true)
 |-- CustomerID: double (nullable = true)
 |-- Country: string (nullable = true)
```

Booleans

- You can build logical statements to evaluate either *true* or *false*
  - `from pyspark.sql.functions import col`
  - `df.where(col("InvoiceNo") != 536365).select("InvoiceNo",`
- A cleaner solution would be to specify a predicate as an expression in a string
  - Another way to say *does not equal*
  - `df.where("InvoiceNo <> 536365").show(5, False)`
- In Spark, you should always chain together and filers
  - Even if filters are expressed serially, Spark will flatten all of these filters into one statement and performed the filter at the same time

```
from pyspark.sql.functions import instr
priceFilter = col("UnitPrice") > 600
descripFilter = instr(df.Description, "POSTAGE") >= 1
df.where(df.StockCode.isin("DOT")).where(priceFilter | desc
```

```
SELECT * FROM dfTable WHERE StockCode in ("DOT") AND (UnitP
```

Boolean Column

```
from pyspark.sql.functions import instr
DOTCodeFilter = col("StockCode") == "DOT"
priceFilter = col("UnitPrice") > 600
descripFilter = instr(col("Description"), "POSTAGE") >= 1
df.withColumn("isExpensive", DOTCodeFilter & (priceFilter

--in SQL
SELECT UnitPrice, (StockCode = 'DOT' AND (UnitPrice > 600 o
FROM dfTable
WHERE (StockCode = 'DOT' AND (UnitPrice > 600 OR instr(Desc
```

Working With Numbers and Nulls

- When using comparisons, watch out for Nulls
  - There is a way to do a null-safe comparison
  - `df.where(col("Description").eqNullSafe("hello")).show()`
- Working with Big Data, after you filter things (WHERE clause), the next task is to count things
- We are able to do math on numeric typed fields, such as:
  - Exponents 82
  - Addition and subtraction
  - All of these features are available in SQL as well and can be written as *selectExpr*
  - Rounding up by default, rounding down, truncating

Basic ANOVA

- Usually there a basic set of statistical methods that are always useful
  - Calculate the Pearson Correlation
  - `from pyspark.sql.functions import corr`
  - `df.stat.corr("Quantity", "UnitPrice")`
  - `df.select(corr("Quantity", "UnitPrice")).show()`
- You can use a single `describe()` method to generate the following:
  - count
  - mean
  - Standard deviation
  - min and max
  - `df.describe().show()`
- These aggregations can be preformed by importing the individual libraries
- Additional statistical functions are available in the **StatFunctions** Package
  - Python StatFunction Library Reference

Working With Strings

- String manipulation is important because you might not have any guarantees about the data you are working with
  - What if the data when entered didn't have any capitalization enforcement?
  - How does this affect string comparisons?
  - `from pyspark.sql.functions import initcap`
  - `df.select(initcap(col("Description"))).show()`
  - `lower` and `upper` will cast entire strings
  - Trimming or padding whitespace 85
  - Regex (Regular Expressions)
  - Use the `contains` method on a column to return a `true` or `false`
  - Python uses the `instr` function in place of `contains`

Working With Dates and Timestamps

- Always challenging because they are numbers but not ordinal
  - Sometimes dates are stored as strings because of this
  - Timezones and Daylight Saving time
  - Even though Spark uses Java dates and timestamps it only works to second level precision
  - No milliseconds
  - Java does have support for date addition and subtraction 91
  - Can also do date comparison and difference between two dates (number of days)
  - `to_date` function allows for converting a string to a date (format can be specified)

Working With Nulls

- Nulls can be represent missing or empty data
    - Spark is internally optimized for use of nulls as opposed to empty strings or zeros
    - But does using *nulls* make sense in your logic
    - Spark has built in functions to check for nulls
    - The `df.na.drop("any")` can be used to drop any row in which any value is null
    - `fill()` and `replace()`

Conclusion

- This chapter demonstrated how easy it is to extend Spark SQL and in a way that is easy to understand.

Questions

- Any questions?
- Read Chapter 07 and do any exercises in the book.