Introduction to Pyspark
with good data engineering practices

by Oliver Willekens,
data engineer and instructor
at Data Minded

dataminded

Course taught online for KBC, Belgium
between 2020-10-12 and 2020-10-23

Introductions

1. **your full name**

   Oliver Willekens

2. **your background (keep it high level) (e.g. “I have a background in social sciences”)**

   Physics engineering.

3. **number of years you've been using Python**

   About 9. Four of those with Spark.

4. **What do you hope to get out of this training? Why are you here?**

   I’m here to help you. Teach tricks. Introduce software engineering practices.

5. **A specific question or problem you would like to see addressed.**

   ~~Finding the sweet spot between the advanced/intermediate users and the starters.~~ → entry tests

   Finding a good way of working for remote teaching with small groups.

Today’s agenda

- Theory
  - Hadoop
  - Spark
    - Spark Stack
    - Spark inter process communication
    - The DataFrame API
- Practice
  - Working with virtual environments and Pycharm

Hadoop is an ecosystem designed to deal with data across cluster nodes. It is built on top of 4 components.

"Ecosystem" is pretty apt:

- Hadoop Common
- Hadoop Distributed File System (HDFS)
- Hadoop YARN
- Hadoop MapReduce

Fun fat: Hadoop got its name from one of the main developers's son. The two year old had a stuffed animal - a yellow elephant, which he called Hadoop.

Doug Cutting, with "Hadoop"

The main concepts behind Hadoop MapReduce can be explained with a deck of cards

Classroom Experiment: need 2 volunteers and a shuffled deck of cards.

Simulate the computation of finding the largest card value per suit, assuming that non-numbered cards are “bad”.

Explain terms like node, process, shuffle, map and reduce. Master/worker.

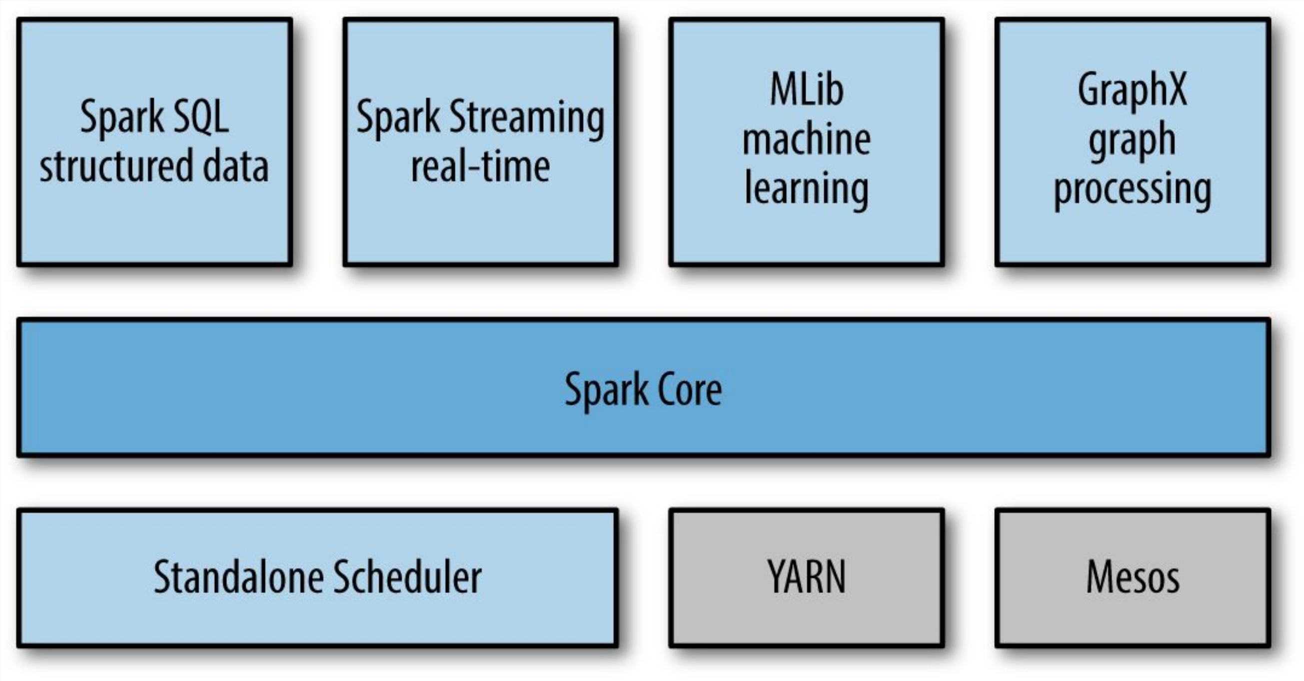
Apache Spark does not replace all of Hadoop. Instead, it replaces Hadoop MapReduce. It integrates well with YARN and HDFS.

- Hadoop Common
- Hadoop Distributed File System (HDFS)
- Hadoop YARN
- Hadoop MapReduce

The Spark Stack consists of 4 modules, one common component and a set of operators that allow integrating with resource managers

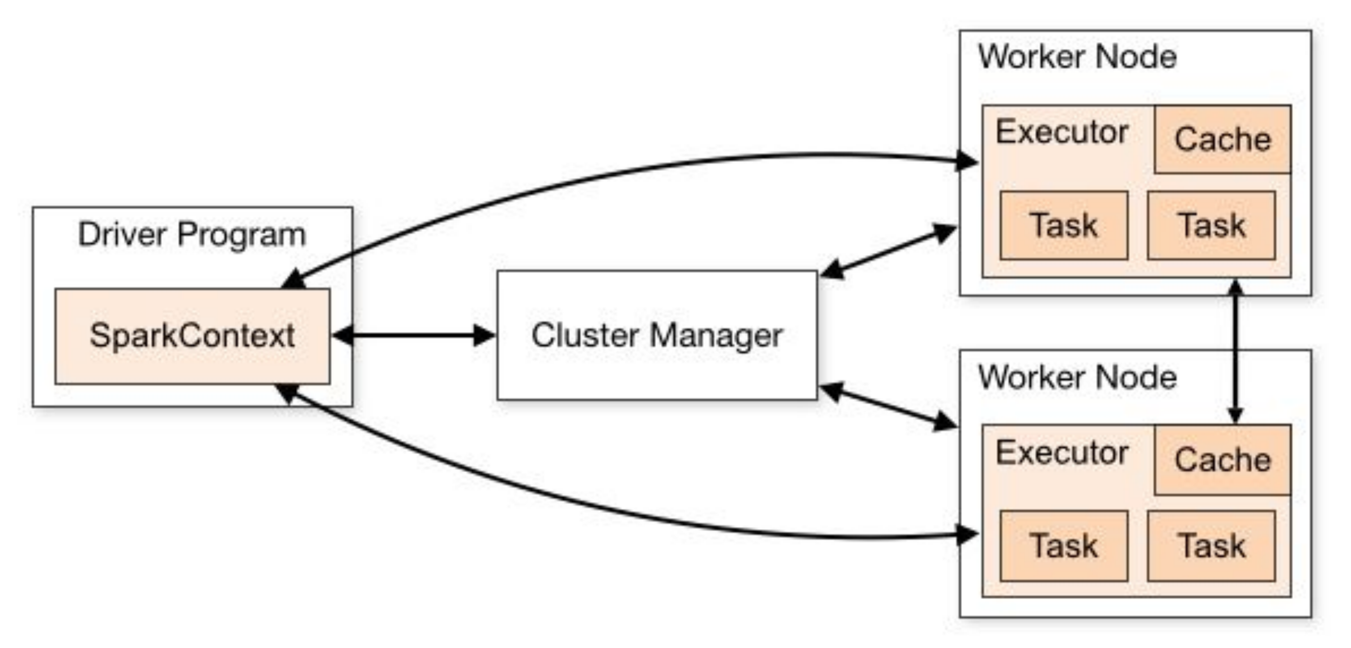
The Spark Stack consists of 4 modules, one common component and a set of operators that allow integrating with resource managers

Cluster modes →



+ kubernetes

Communication between components in a Spark application happens by all actors



Which edge in this diagram has not been discussed? Can you come up with a reason for its existence?

How does High Performance Computing differ from Spark/MapReduce?

Core concepts of the Spark API

- RDDs
- Datasets
- Row
- Column
- SparkSession

Demos with a pyspark-shell

Production-grade code comes with tests. They allow you to change code in the future, with a feeling of assuredness that stuff still works.

The rationale behind tests:

- Improves chance of code still being correct in the *future*
    - Code likely works now: people have the tendency to test their code (manually) on a small problem
    - Code will change, as requirements and environments change.
    - To prevent introducing breaking changes: write tests and ship these with the code.
- Raises confidence that code is correct now
    - assert that the results match expectations
    - trains you to think about edge cases, which aren't so uncommon as people may believe. Programming is an art about details. This is often times why non-techies do not understand that coding something up properly, can take a while.
- Most up-to-date form of documentation
    - word documents and wikis will grow out of sync with the code.
    - tests usually target a very specific piece of functionality and help you reason about those pieces in the bigger picture

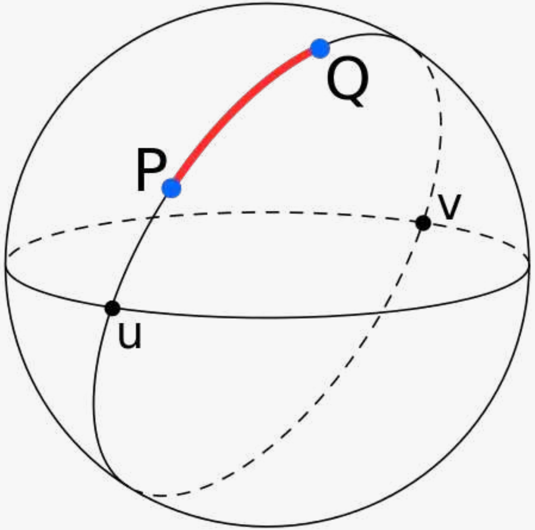
Pytest is one of the most well-known testing libraries in the Python ecosystem

Alternatives: unittest, doctest, nose

A basic test **asserts** something:

statement evaluating to bool	meaning
2 != 3	the numerical value of 2 is not that of 3
len(“hello”) == len(“world”)	the strings “hello” and “world” have the same number of characters
{1, 2, 3}.issubset(range(5))	the former set is a subset of the latter collection

A warm-up to testing PySpark code: let’s write a unit test for the great-circle-distance metric!



The great-circle-distance (gcd) or Haversine distance gives the shortest distance along the surface of a sphere between any two points.

It is a commonly encountered problem in anything related to locations.

Agenda for session 2

- **Actions** vs **transformations** in Spark
- Warm-up exercise: **label holidays**
- Writing PySpark unit tests with Pytest
  - configuring your IDE
  - building the **helper functions**
- Performance considerations of **User-defined functions** (UDFs)
  - how-to write UDFs
  - example: label holidays, 3 ways!

Little recap of the PySpark functionality you should be familiar with

```
spark = SparkSession.builder.getOrCreate()

df = spark.createDataFrame(
   data=[row1, row2, …, rowN],
   schema=sequence_of_column_names_or_a_structtype
)

df.select("id").withColumn("foo", psf.upper(col("bar"))).orderBy("foo").show()

spark.read.csv(some_path, **options).write.repartition(N).parquet(some_other_path)
```

```
# for better readability:
foo = (
    spark
    .read
    .csv(some_path, **options)
    .write
    .repartition(N)
    .parquet(some_other_path)
)
```

Humans read quicker from top to bottom, as it’s a scanning operation. We tend to give less attention to stuff on the right side, as it’s considered to be details.

Do let a linter reformat your code though.

Pop quiz: action or transformation

df = spark.range(0)

other_df = spark.range(0)

df.withColumn(“foo”, lit(5))	transformation
df.join(other_df, on=[“id”], how=”left”)	transformation
df.count	neither: simply a ref to a bound method
df.count()	action
df.rdd	neither: an accessor to get to the data in a different way. Close to a transformation though.
df.collect()	action
df.describe()	transformation
df.groupBy(“id”).count()	transformation! Note: this count is a convenience method. It replaces an aggregator.
df.groupBy(“id”).agg(sum(lit(1)).alias(“count”))	transformation: same result as above
df.take(4)	action
df.schema	neither: simply an attribute of a DataFrame
df.cache()	transformation, and just like all others: a lazy one! The DataFrame won’t be cached until the next action.
df.read.parquet(some_file)	tricky one: some work is done, because the file’s metadata is read (e.g. the schema), but the entire file isn’t processed.

The holidays module offers a good playground for commonly used transformations in PySpark, and it’s a common enough request for machine learning applications that it’s good to know about.

The holidays module is on the cheese shop.

Read the Example Usage section of the package.

Next, in *exercises/c_labellers*, extend the function `is_belgian_holiday` so that it can be used in predicates to validate whether a date instance is in fact a Belgian holiday.

Validate your logic using a test (warm-up exercise).

Testing the logic of functions that involve PySpark DataFrames isn’t as easy as a regular unit test, because of the interprocess communication overhead.

Do not:

- create a separate SparkSession per test.
  Instead, start on at the beginning of the test suite and use it throughout. Look into Pytest’s fixtures for session-scoped resources.
- recompute DataFrames by forgetting to cache results (if recomputation would be triggered).

Do:

- create small in-memory DataFrames that illustrate the logic
- write a helper function to validate the **functional equivalence** of two DataFrames.
  Write such a helper function, as a way to make you appreciate the distributed nature of DataFrames. Validate your logic using *tests/test_comparers.py*

User defined functions are a wrapper over pure Python functions. Their typed nature reduces the general character of the original function.

```
from pyspark.sql.functions import udf
from pyspark.sql.types import IntegerType

def square(x):
    return x**2


square_udf_int = udf(lambda z: square(z), IntegerType())

(
    df.select('integers',
              'floats',
              square_udf_int('integers').alias('int_squared'),
              square_udf_int('floats').alias('float_squared'))
    .show()
)
```

desired return type (function should return a Python object, which can be mapped to this type)

function

User defined functions suffer from two drawbacks, making them slow.

```python
from pyspark.sql.functions import udf
from pyspark.sql.types import IntegerType

def square(x):
    return x**2


square_udf_int = udf(lambda z: square(z), IntegerType())

(
    df.select('integers',
              'floats',
              square_udf_int('integers').alias('int_squared'),
              square_udf_int('floats').alias('float_squared'))
    .show()
)
```

A lambda (nameless function) for a function that takes 1 argument when called, is silly, and another layer of overhead.

- high serialization overhead
- large number of invocation calls

User defined functions are not the only way to implement functionality. An incredible amount of work can be done with what Spark provides you.

- The functionality is likely already in [pyspark.sql.functions]()
- You could also look into redesigning the algorithm
- Or go low-level with `DataFrame.rdd.mapPartitions`

We'll see examples of these in the exercises: write functions to label

- weekends
- Belgian holidays

In a PySpark DataFrame.