Introduction to Pyspark

with good data engineering practices

by Oliver Willekens,
data engineer and instructor
at Data Minded

dataminded

Course taught at KBC Leuven, Belgium
between 2019-05-13 and 2019-05-15

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 8. Two and a half 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.

Today's agenda

- Theory
  - Hadoop
  - Spark
    - Spark Stack
    - Spark inter process communication
    - The DataFrame API
- Practice
  - Working with pipenv and Pycharm
  - Writing unit tests with Pytest
  - Testing Spark code locally

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

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

"Ecosystem" is pretty apt:

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

Doug Cutting, with "Hadoop"

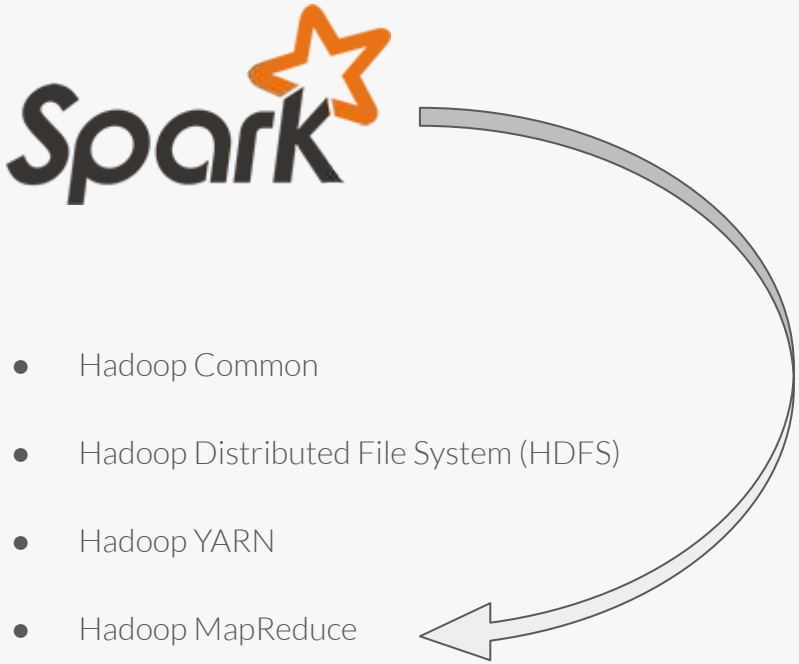
The main concepts behind Hadoop MapReduce can be explained with a deck of cards

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.

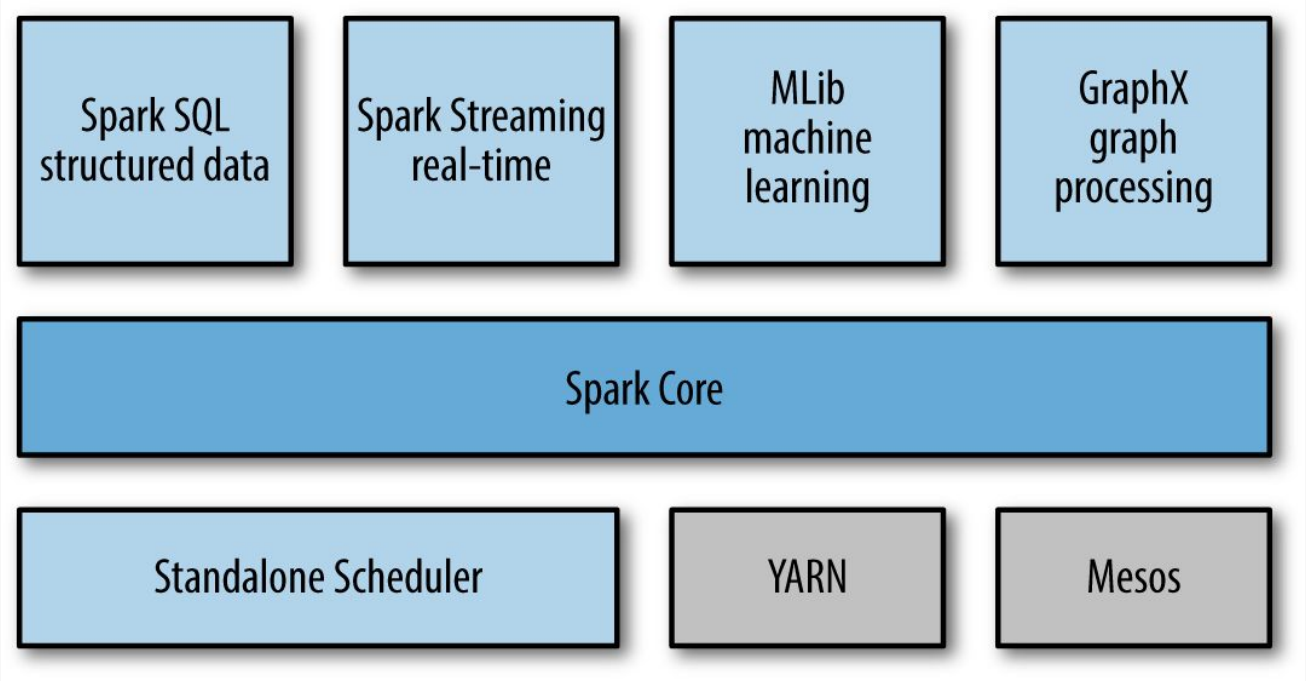
Apache Spark does not replace all of Hadoop. Instead, it replaces Hadoop MapReduce. It integrates well with YARN and HDFS.



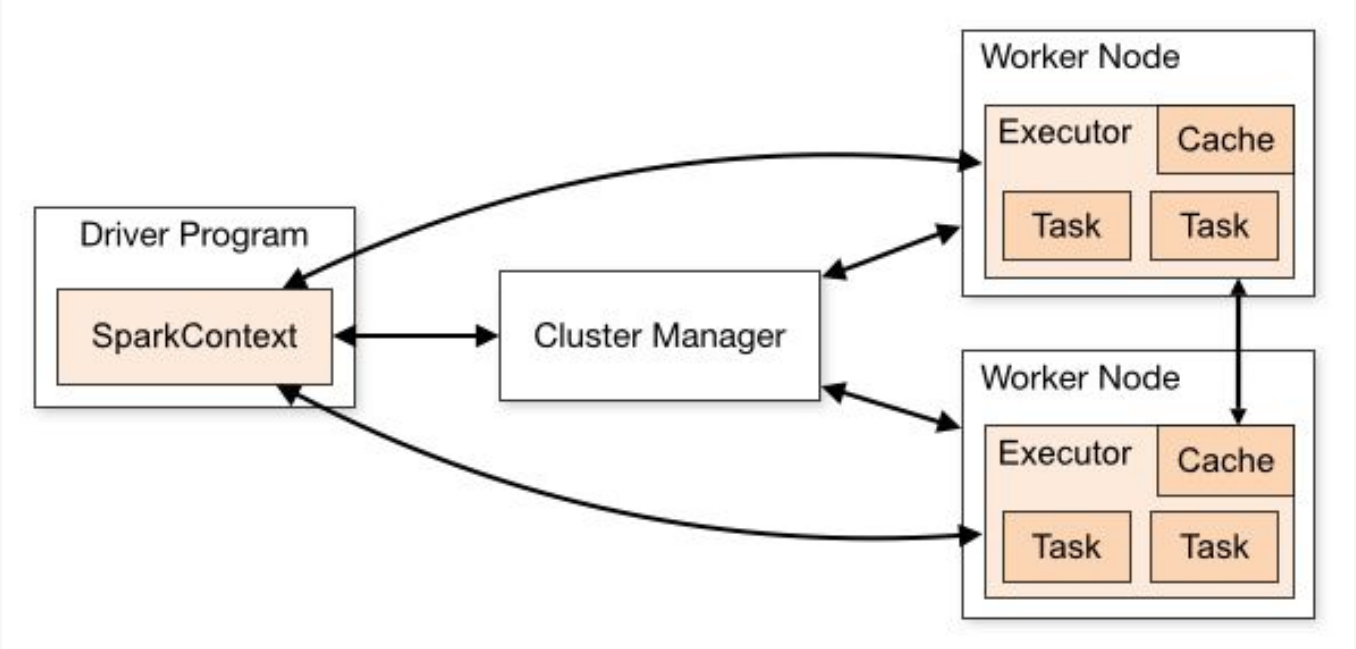
- Hadoop Common
- Hadoop Distributed File System (HDFS)
- Hadoop YARN
- Hadoop MapReduce

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



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

Agenda for day 2

- Performance considerations of User-defined functions

- Unit test: label holidays
- Analysis of a query plan
- Data serialization formats: avro, parquet & orc

- Use case 1: “cleaning” a CSV file of flights

User defined functions are a wrapper over pure Python functions

```
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()
)
```

User defined functions suffer from two drawbacks

```
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 for a function that takes 1 argument when called, is silly

- 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.mapPartitions()`

We'll see examples of these in the exercises.

There are 3 widely used file formats for big data: Avro, Parquet and ORC

		
Avro is a row-oriented file format, so better in streaming cases or for operational datastores	Apache Parquet is a columnar storage format: great for analytics.	ORC, short for Optimized Row Column format is similar to Parquet in its uses. Support for it is (at the time of writing) smaller than Apache Parquet.

Agenda for day 3

- The data catalog
  - Notions
  - Implementation
- Code review
- Implementing a business use case: revisiting the (cleaned) airline dataset

But first we pick up where we left off

We missed a few interesting points yesterday.

1. file sizes of the pre-clean and post-clean data (+repartition per user)
2. illustrate caching in the Spark UI (+unpersist)
    a. when to use
    b. how to observe
3. partitioning scheme on write
    a. repartition
    b. partitionBy

At its core, a data catalog is an object holding metadata: information about your data.

Metadata examples:

- storage location
- format
  - Parquet
  - SQL → JDBC/ODBC
  - JSON
  - ...
- meaningful aggregation statistics
  - size/number of records
  - distribution of the data (min, max, nullcount)

Code review

With everything you've learned so far, you should be able to read, understand and improve someone else's work.

We will step through an actual use case of a different business: finding out which people are "bingewatchers". Note that we do not need to understand the criterion to define when someone is flagged as a bingewatcher. But we can still improve this code.

Exercise: implement a simple data catalog

Implement a data catalog using a dictionary.
Put the data catalog in its own module, as it is important enough by itself. In the catalog, refer to the following files:

1. the flights data of the year 2008, which we've cleaned in the previous session.
2. The lookup table that maps IANA airport codes to their full names
3. The lookup table that maps IATA carrier codes to airline carriers over a specific timerange.

Build up the catalog gradually:

1. link a dataset name to a location (bonus: make it independent of the location of your project on the laptop)
2. add sufficient information about the format so that it can be used by an automated process
3. add format-specific options. Again, see that it can be used in an automated way.
4. Create a *reader* function that, given a key of the catalog, will return the corresponding dataframe.

Exercise: implementing a business use case. Pretend you work for American Airlines

- Create a 360°-view of the flights data in which you combine the airline carriers (a dimension table), the airport names (another dimension table) and the flights tables (a facts table).

- Your manager wants to know how many flights we operated in 2008.
- How many of those flights arrived with less than 10 minutes of delay?

- A data scientist is looking for correlations between the departuredelays and the dates. In particular, he/she thinks that on Fridays there are relatively speaking more flights departing with a delay than on any other day of the week. Verify his/her claim.
- Out of the 4 categories of sources for delays, which one appeared most often in 2008? In other words, in which category should we invest more time to improve?
- In 2008, which 3 airlines made the longest flights? (not made in class, try at home)
- Which airlines no longer exist? (not made in class, try at home)