> **Algoritmos para Big Data**
>
> **2023/2024**

Introduction to Apache Spark

In this lecture we will introduce the Spark framework. For the time being, the goal is to explain how it works and to highlight its potentiality.

Disclaimer: Some content presented in this notebook e.g. images are based on references mentioned at the end of the notebook.

Context

Long time ago, computers got faster mainly due to processor speed increases. And most of the applications were designed to run in a single processor machine. But as more data was required to be processed and hardware limits were being tested, research efforts moved towards parallel processing and new programming models.

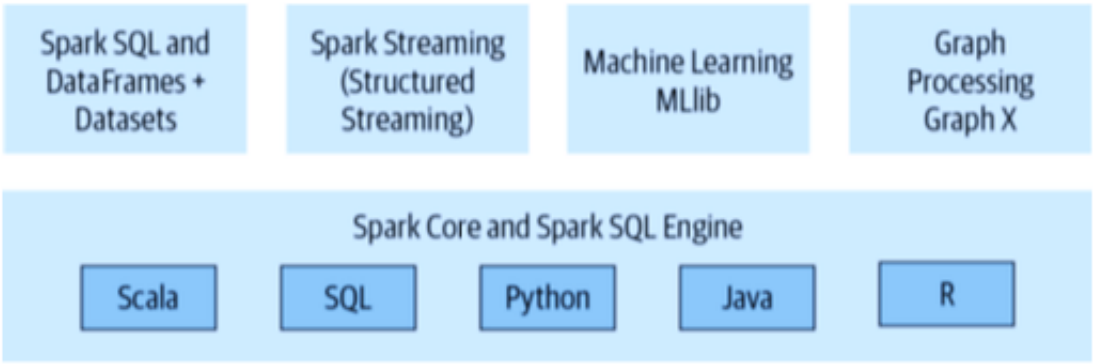
Apache Spark

- is an open-source distributed cluster-computing framework. It is designed for large-scale distributed data processing, with focus on speed and modularity;
- provides in-memory storage for intermediate computations;
- contain libraries with APIs for machine learning, SQL, stream processing and graph processing.

Spark components and APIs

Spark offers four components as libraries for diverse workloads in a unified stack.

Code can be written in the languages Scala, SQL, Python, Java or R, which then is decomposed into bytecode to be executed in Java Virtual Machines (JVMs) across the cluster.



There are both low-level and high-level APIs related to (distributed) collections of data. We may have collections of:

- **Resilient Distributed Dataset (RDD)**
  - they are now consigned to low-level APIs
- **DataFrame**
  - the most commom structured data - it simply represents a table of data with rows and columns
- **Dataset**
  - collection of objects but only makes sense in the case of Scala and Java

Further details are to be covered later on but we can highlight now that our focus will be on **DataFrame**

Spark Core and Spark SQL Engine

Spark Core contains basic functionalities for running jobs and that are needed by other components. Spark SQL Engine provides additional help to do so.

Computations will ultimatelly convert into low-level RDD-based bytecode (in Scala) to be distributed and run in executors across the cluster.

Spark SQL

Spark SQL provides functions for manipulating large sets of distributed structured data using an SQL subset. (ANSI SQL:2003-compliant)

It can also be used for **reading** and **writing** data to and from various structured formats and data sources, such as JavaScript Object Notation (JSON) files, CSV files, Parquet files (an increasingly popular file format that allows for storing a schema alongside the data), relational databases, Hive, and others.

There is also a query optimization framework called Catalyst.

Spark Structured Streaming

Spark Structured Streaming is a framework for ingesting real-time streaming data from various sources, such as HDFS-based, Kafka, Flume, Twitter, ZeroMQ, as well as customized ones.

Developers are able to combine and react in real time to both static and streaming data. A stream is perceived as a continuaslly growing structured table, upon against which queries are made as if it was a static table.

Aspects of fault tolerance and late-data semantics are handled via Spark SQL core engine. Hence, developers are focussing on just writing streaming applications.

Machine Learning MLlib

Spark MLlib is a library of common machine-learning (ML) algorithms built on top of DataFrame-based APIs. Among other aspects, these APIs allow to extract or transform features, build pipelines (for training and evaluating) and persist models during deployment (for saving/reloading)

Available ML algorithms include logistic regression, naïve Bayes classification, support vector machines (SVMs), decision trees, random forests, linear regression, k-means clustering, among others.

Graph Processing GraphX

Spark Graphx is a library for manipulating graphs, that is, data structures comprising vertices and the edges connecting them.

It provides algorithms for building, analysing, connecting and traversing graphs. Among others, there are implementations of important algorithms of graph theory, such as page rank, connected components, shortest paths and singular value decomposition. (SVD)

Application running

Conceptually, we prototype an application by running it locally with small datasets; then, for large datasets, we use more advanced deployment modes to take advantage of distributed and more powerful execution.

As far as the running of the application is concerned, it can run interactively - either via shells (Terminal windows) or notebooks - or as a standalone application.

Spark provides four interpretative shells to carried out ad-hoc data analysis:

- pyspark
- spark-shell
- spark-sql
- sparkR

They resemble their shell counterparts for the considered languages. The main difference now is that they have extra support for connecting to the cluster and to loading distributed data into worker's memory.

Notice that, if using shells:

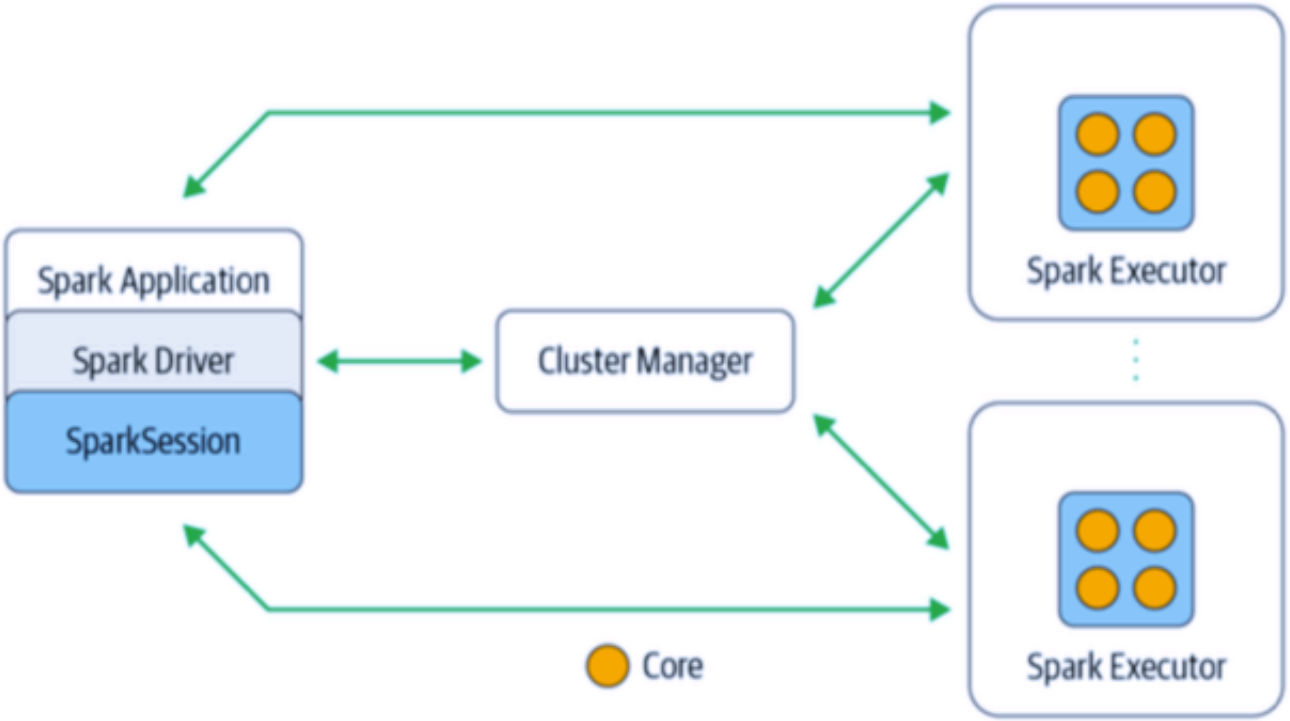
- the driver is part of the shell
- the SparkSession mentioned above is automatically created, accessible via the variable `spark`
- they are exited pressing Ctrl-D

In the case of a standalone application, the code is written in an Integrated Development Environment (IDE) and then is deployed to run after the command `spark-submit`.

We will use Spark interactively through Jupyter notebooks.

Execution in a distributed architecture

A **Spark Application** consists of a **driver** program responsible for orchestrating parallel operations on the Spark cluster. The driver accesses the distributed components in the cluster (**executors** and **manager**) via a **SparkSession**.



SparkSession

A SparkSession instance provides a single entry point to all functionalities, either by running as standalone aplication, or by running in the notebook.

For a Spark application, one needs to create the SparkSession object if none is available, as described below. In that case, we can configure it according to ower own needs.

But first, we have to make sure we can access Spark from this notebook. One way to do so is to run the notebook using a suitable kernel. That is why we have already set one, the current `pyspark_env` kernel.

In [1]:
```
import findspark
findspark.init()
findspark.find()

import pyspark
from pyspark.sql import SparkSession

# build our SparkSession

spark = SparkSession\
        .builder\
        .appName("BigData")\
        .config("spark.sql.shuffle.partitions",6)\
        .config("spark.sql.repl.eagereval.enabled",True)\
        .getOrCreate()
```

```
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
24/02/19 13:15:56 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-
```

In [2]:
```
# Check it, incluing the link Spark UI
```

Out[2]: **SparkSession - in-memory**

SparkContext

Spark UI

Version	`v3.5.0`
Master	`local[*]`
AppName	`BigData`

Cluster manager and executors

- **Cluster manager**
  - responsible for managing the executors in the cluster of nodes on which the application runs, alongside allocating the requested resources
  - agnostic where ir runs as long as responsabilities above are met
- **Spark executor**
  - runs on each worker node in the cluster
  - executors communicate with driver program and are responsible for executing tasks on the workers
- **Deployment modes**
  - variety of configurations and environments available, as shown below: (just for reference)

Mode	Spark driver	Spark executor	Cluster manager
Local	Runs on a single JVM, like a laptop or single node	Runs on the same JVM as the driver	Runs on the same host
Standalone	Can run on any node in the cluster	Each node in the cluster will launch its own executor	Can be allocated arbitrarily to any host in the cluster
YARN (client)	Runs on a client, not part of the cluster	YARN's NodeManager's container	YARN's Resource Manager works with YARN's Application Master to allocate the containers on NodeManagers for executors
YARN (cluster)	Runs with the YARN Application Master	Same as YARN client mode	Same as YARN client mode
Kubernetes	Runs in a Kubernetes pod	Each worker runs within its own pod	Kubernetes Master

Distributed data and partitions

- Partitioning of data allows for efficient paralelism since every executor can perform work in parallel
- Physical data is break up and distributed across storage as chunks called partitions, whether in HDFS or in cloud storage.
- Each partition is treated as a dataframe in memory (logical data abstraction)
  - hence it is a collection of rows that sits on one physical machine of the cluster;
  - so if we have dataframes in our program we do not (for the most part) manipulate partitions individually - we simply specify high level transformations of data in the physical partitions, and Spark determines how this will play out across the cluster.
- As much as possible, data locality is to be pursuit. It means that an executor is prefereably allocated a task that requires reading a partition closest to it in the network in order to minimize network bandwidth.

Question: What happens if we have

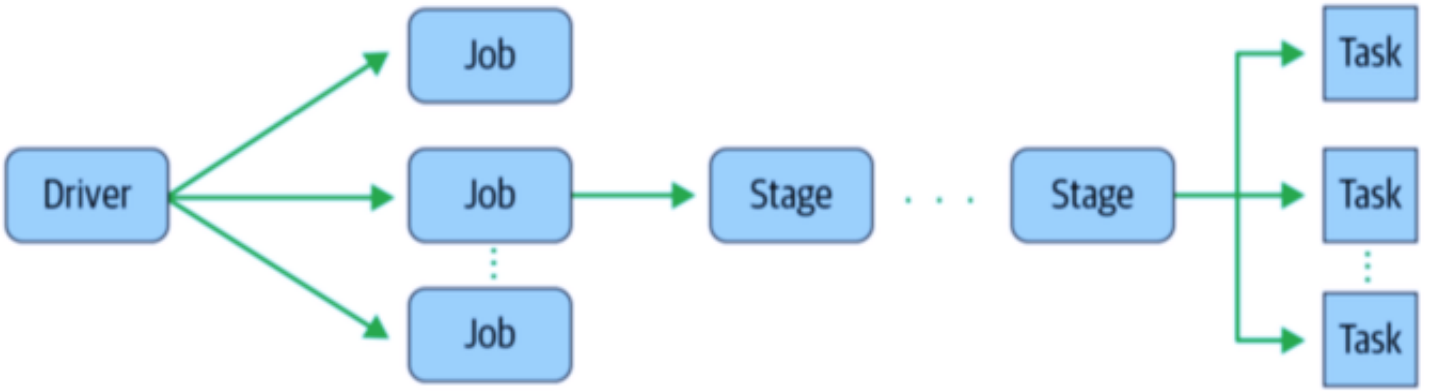
- multiple partitions but only one executor;
- one partition but thousands of executors?

Spark operations and related computation

Operations on distributed data are of two types: **transformations** and **actions**

Basic concepts

- **Job**: parallel computation created by the driver, consisting of multiple tasks that gets spawned in response to actions, e.g save()
- **Stage**: each job gets divided into smaller sets of tasks called stages, that depend on each other
- **Task**: single unit of work or execution to be sent to a Spark executor (a task per core)



Transformations

In Spark core data structures are **Immutable**, that is, they cannot be changed after creation. If one wants to change a dataframe we need to instruct Spark how to do it. These are called transformations.

Hence, transformations transform a DataFrame into a new one without altering the original data. So it returns a new one but transformed.

Some examples are:

Transformation	Description
orderBy()	Returns a new DataFrame sorted by specific column(s)
groupBy()	Groups the DataFrame using specified columns, so we can run aggregation on them
filter()	Filters rows using a given condition
select()	Returns a new DataFrame with select columns
join()	Joins with another DataFrame, using a given join expression

These transformations will be explained in more detailed in the next lecture.

Let us apply some commands to the contents of the directory this notebook is located.

In [ ]:
```
# Check the current dir and its content via - these are Linux commands, e.g. pwd, ls

! pwd
! ls -la
! ls -la > mydir.txt
```

In [ ]:
```
# Reading a text file
# strings =

strings = spark.read.
strings.count()
```

In [5]:
```
# See our dataframe schema

strings.printSchema()
```

```
root
 |-- value: string (nullable = true)
```

In [ ]:
```
# Show only 5 full lines (not truncated)

strings.show(5, truncate=False)
```

In [ ]:
```
# Filtering lines with a particular word (example: "Spark" or "staff")

filtered = strings.
filtered.show(truncate=False)
filtered.count()
```

Types of transformations

Transformations can be:

- Narrow
  - a single output partition can be computed from a single input partition (no exchange of data, all performed in memory)
  - examples are **filter()**, **contains()**
- Wide
  - data from other partitions across the cluster is read in, combined, and written to disk
  - examples are **groupBy()**, **reduceBy()**

Example

Reading structured data, filter some of them and then show the result but sorted

(The file is on the same folder as this notebook)

In [ ]:
```
# Prior, just let us check the file we are about to use (with help of Linux commands)

! ls -la
```

In [ ]:
```
! head ./us-airline-delay-2016-2018/2018.csv
```

In [10]:
```
# Read the datafile into a DataFrame using the CSV format,
# by inferring the schema and specifying that the file contains a header,
# which provides column names for comma-separated fields

file_path = "./us-airline-delay-2016-2018/2018.csv"
flights = spark.read.load(file_path,
                          format="csv",
                          sep=",",
                          inferSchema="true",
                          header="true")
flights.printSchema()
flights # or print(flights)
```

```
[Stage 9:==============>                                          (2 + 6) / 8]
```

```
root
 |-- FL_DATE: date (nullable = true)
 |-- OP_CARRIER: string (nullable = true)
 |-- OP_CARRIER_FL_NUM: integer (nullable = true)
 |-- ORIGIN: string (nullable = true)
 |-- DEST: string (nullable = true)
 |-- CRS_DEP_TIME: integer (nullable = true)
 |-- DEP_TIME: double (nullable = true)
 |-- DEP_DELAY: double (nullable = true)
 |-- TAXI_OUT: double (nullable = true)
 |-- WHEELS_OFF: double (nullable = true)
 |-- WHEELS_ON: double (nullable = true)
 |-- TAXI_IN: double (nullable = true)
 |-- CRS_ARR_TIME: integer (nullable = true)
 |-- ARR_TIME: double (nullable = true)
 |-- ARR_DELAY: double (nullable = true)
 |-- CANCELLED: double (nullable = true)
 |-- CANCELLATION_CODE: string (nullable = true)
 |-- DIVERTED: double (nullable = true)
 |-- CRS_ELAPSED_TIME: double (nullable = true)
 |-- ACTUAL_ELAPSED_TIME: double (nullable = true)
 |-- AIR_TIME: double (nullable = true)
 |-- DISTANCE: double (nullable = true)
 |-- CARRIER_DELAY: double (nullable = true)
 |-- WEATHER_DELAY: double (nullable = true)
 |-- NAS_DELAY: double (nullable = true)
 |-- SECURITY_DELAY: double (nullable = true)
 |-- LATE_AIRCRAFT_DELAY: double (nullable = true)
 |-- Unnamed: 27: string (nullable = true)
```

Out[10]:
```
DataFrame[FL_DATE: date, OP_CARRIER: string, OP_CARRIER_FL_NUM: int, ORIGIN: string, DEST: string, CRS_DEP_TIME:
```

In [11]:
```
# Check how many records we have in the data frame

flights.
```

Out[11]: 7213446

In [12]:
```
# and showing some of them ex: 2 (consider the option vertical=True)

#flights.show(2,
flights.show(
```

```
+----------+----------+-----------------+------+----+------------+--------+---------+--------+----------+--------
|FL_DATE   |OP_CARRIER|OP_CARRIER_FL_NUM|ORIGIN|DEST|CRS_DEP_TIME|DEP_TIME|DEP_DELAY|TAXI_OUT|WHEELS_OFF|WHEELS_O
+----------+----------+-----------------+------+----+------------+--------+---------+--------+----------+--------
|2018-01-01|UA        |2429             |EWR   |DEN |1517        |1512.0  |-5.0     |15.0    |1527.0    |1712.0
|2018-01-01|UA        |2427             |LAS   |SFO |1115        |1107.0  |-8.0     |11.0    |1118.0    |1223.0
|2018-01-01|UA        |2426             |SNA   |DEN |1335        |1330.0  |-5.0     |15.0    |1345.0    |1631.0
|2018-01-01|UA        |2425             |RSW   |ORD |1546        |1552.0  |6.0      |19.0    |1611.0    |1748.0
|2018-01-01|UA        |2424             |ORD   |ALB |630         |650.0   |20.0     |13.0    |703.0     |926.0
|2018-01-01|UA        |2422             |ORD   |OMA |2241        |2244.0  |3.0      |15.0    |2259.0    |1.0
|2018-01-01|UA        |2421             |IAH   |LAS |750         |747.0   |-3.0     |14.0    |801.0     |854.0
|2018-01-01|UA        |2420             |DEN   |CID |1324        |1318.0  |-6.0     |11.0    |1329.0    |1554.0
|2018-01-01|UA        |2419             |SMF   |EWR |2224        |2237.0  |13.0     |10.0    |2247.0    |627.0
|2018-01-01|UA        |2418             |RIC   |DEN |1601        |1559.0  |-2.0     |12.0    |1611.0    |1748.0
+----------+----------+-----------------+------+----+------------+--------+---------+--------+----------+--------
only showing top 10 rows
```

```
24/02/19 13:16:08 WARN SparkStringUtils: Truncated the string representation of a plan since it was too large. Th
```

In [13]:
```
# Some of the columns are not interesting. Let us get rid of them. Feel free to delete more.
# Also, because dataframes are big and consume memory, we are going to delete the one
# that is no longer needed

cols_to_drop = ["TAXI_OUT", "TAXI_IN", "WHEELS_OFF", "WHEELS_ON",
                "CARRIER_DELAY", "WEATHER_DELAY",
                "NAS_DELAY", "SECURITY_DELAY",
                "LATE_AIRCRAFT_DELAY", "Unnamed: 27"]
info_flights = flights.drop(*cols_to_drop)

del flights
```

In [14]:
```
# Show the flights

info_flights.show(2, vertical=False, truncate=False)
```

```
+----------+----------+-----------------+------+----+------------+--------+---------+------------+--------+------
|FL_DATE   |OP_CARRIER|OP_CARRIER_FL_NUM|ORIGIN|DEST|CRS_DEP_TIME|DEP_TIME|DEP_DELAY|CRS_ARR_TIME|ARR_TIME|ARR_DE
+----------+----------+-----------------+------+----+------------+--------+---------+------------+--------+------
|2018-01-01|UA        |2429             |EWR   |DEN |1517        |1512.0  |-5.0     |1745        |1722.0  |-23.0
|2018-01-01|UA        |2427             |LAS   |SFO |1115        |1107.0  |-8.0     |1254        |1230.0  |-24.0
+----------+----------+-----------------+------+----+------------+--------+---------+------------+--------+------
only showing top 2 rows
```

> The normal show of data frames is not pretty. We can improve it if we set first the html display as below. Try again!

In [15]:
```
from IPython.core.display import HTML
display(HTML("<style>pre { white-space: pre !important; }</style>"))
```

Lazy evaluation and actions

Spark uses lazy evaluation, that is, it waits until the very last moment to execute the graph of computational instructions established, that is, the plan of transformations that we would like to apply to the data.

As results are not computed immediately, they are recorded as **lineage** (*trace of descendants*) and at later time in its execution plan, Spark may rearrange certain transformations, coalesce them, or optimize transformations into stages for more efficient execution of the entire flow. Hence, far from laziness, this makes it faster actually.

Only when an **action** is invoked or data is read/written to disk the lazy evaluation of all recorded transformations is triggered.

An action is like a play button. We may have:

- Actions to view data in the console.
- Actions to collect data to native objects in the respective language.
- Actions to write to output data sources.

Some examples are:

Action	Description
show()	Prints the first rows to the console
take()	Returns the first rows as a list
count()	Returns the number of rows
collect()	Returns all the records as a list
save()	Saves the contents to a data source

In [16]:
```
# Using the variable info_flights (a DataFrame) set before...

selected_flights = info_flights.where(
# why didn't return the output, in this case regarding only United Airline flights?
```

In [17]:
```
# How is going to compute the result?

selected_flights.explain() # or selected_flights.explain(extended=True)
```

```
== Physical Plan ==
*(1) Filter (isnotnull(OP_CARRIER#46) AND (OP_CARRIER#46 = UA))
+- FileScan csv [FL_DATE#45,OP_CARRIER#46,OP_CARRIER_FL_NUM#47,ORIGIN#48,DEST#49,CRS_DEP_TIME#50,DEP_TIME#51,DEP_
```

In [18]:
```
# Count

selected_flights.
```

Out[18]:
```
621565
```

In [19]:
```
# Get 5 of them

selected_flights.take(5)
```

Out[19]:
```
[Row(FL_DATE=datetime.date(2018, 1, 1), OP_CARRIER='UA', OP_CARRIER_FL_NUM=2429, ORIGIN='EWR', DEST='DEN', CRS_DE
 Row(FL_DATE=datetime.date(2018, 1, 1), OP_CARRIER='UA', OP_CARRIER_FL_NUM=2427, ORIGIN='LAS', DEST='SFO', CRS_DE
 Row(FL_DATE=datetime.date(2018, 1, 1), OP_CARRIER='UA', OP_CARRIER_FL_NUM=2426, ORIGIN='SNA', DEST='DEN', CRS_DE
 Row(FL_DATE=datetime.date(2018, 1, 1), OP_CARRIER='UA', OP_CARRIER_FL_NUM=2425, ORIGIN='RSW', DEST='ORD', CRS_DE
 Row(FL_DATE=datetime.date(2018, 1, 1), OP_CARRIER='UA', OP_CARRIER_FL_NUM=2424, ORIGIN='ORD', DEST='ALB', CRS_DE
```

In [20]:
```
# The 1st one

selected_flights.
```

Out[20]:
```
Row(FL_DATE=datetime.date(2018, 1, 1), OP_CARRIER='UA', OP_CARRIER_FL_NUM=2429, ORIGIN='EWR', DEST='DEN', CRS_DEP
```

In [21]:
```
# and show

selected_flights.show(5, truncate=False)
```

```
+----------+----------+-----------------+------+----+------------+--------+---------+------------+--------+------
|FL_DATE   |OP_CARRIER|OP_CARRIER_FL_NUM|ORIGIN|DEST|CRS_DEP_TIME|DEP_TIME|DEP_DELAY|CRS_ARR_TIME|ARR_TIME|ARR_DE
+----------+----------+-----------------+------+----+------------+--------+---------+------------+--------+------
|2018-01-01|UA        |2429             |EWR   |DEN |1517        |1512.0  |-5.0     |1745        |1722.0  |-23.0
|2018-01-01|UA        |2427             |LAS   |SFO |1115        |1107.0  |-8.0     |1254        |1230.0  |-24.0
|2018-01-01|UA        |2426             |SNA   |DEN |1335        |1330.0  |-5.0     |1649        |1636.0  |-13.0
|2018-01-01|UA        |2425             |RSW   |ORD |1546        |1552.0  |6.0      |1756        |1754.0  |-2.0
|2018-01-01|UA        |2424             |ORD   |ALB |630         |650.0   |20.0     |922         |936.0   |14.0
+----------+----------+-----------------+------+----+------------+--------+---------+------------+--------+------
only showing top 5 rows
```

Fault tolerance

Lineage in the context of lazy evaluation and **data immutability** mentioned above gives resiliency in the event of failures as:

- Spark records each transformation in its lineage;
- DataFrames are immutable between transformations;

then Spark can reproduce the original state by replaying the recorded lineage.

Spark UI

Spark UI allow us to monitor the progress of a job. It displays information about the state of Spark jobs, its environment and the cluster state. So it is very useful for tuning and debugging.

Usually Spark UI is available on port 4040 of the driver node. (If that port is occupied, another one is provided)

In local mode: http://localhost:4040 in a web browser.

```
PS: recall notebook cell above when spark was checked.
```

In [ ]:
```
# Check where spark ui is running and the link presented after execution

spark.sparkContext.uiWebUrl
```

Exercise

Here we have the introduction to Spark. Now let us further consolidate the concepts with an exercise.

But before, are there drawbacks in using Spark?

Probably yes, as expected. For instance, it there is no much data to process we are better off not using Spark at all. Another aspect is that the management and tunning of the running cluster is not that much user-friendly.

Back to the exercise, our goal now is to write down a Spark program that:

> a) Reads a file containing flight data in 2017.
>
> b) Creates a DataFrame with the columns
>
> ```
> FL_DATE, OP_CARRIER, OP_CARRIER_FL_NUM, ORIGIN, DEST,
> CRS_DEP_TIME, DEP_TIME, DEP_DELAY,
> CRS_ARR_TIME, ARR_TIME, ARR_DELAY,
> CRS_ELAPSED_TIME, ACTUAL_ELAPSED_TIME,
> AIR_TIME, DISTANCE
> ```
>
> and add four more columns with, respectively in relation to the flight date, the year, the month, the day of the week and the week of the year.
>
> c) Provides answers to the following questions about the data under analysis:
>
> 1. How many flights are recorded in the dataset?
> 2. What are the average of departure delay (DEP_DELAY) and the average of arrival delay (ARR_DELAY)?
> 3. Which are the top 3 busiests airpoirts regarding departures?
> 4. Give an airport such as no other airport have more arrivals?
> 5. Which airline holds the heighest number of departures in an airport, and which airport is?

The original dataset can be found via the link

https://www.kaggle.com/datasets/yuanyuwendymu/airline-delay-and-cancellation-data-2009-2018?select=2017.csv

The Spark program

In [23]:
```
# Import necessary libraries

import sys

import pyspark.sql.functions as F   # We will see convenient use of F below
```

In [24]:
```
# Read the dataset

file_path = "./us-airline-delay-2016-2018/2017.csv"

flight_data = ( spark.read.format("csv")
                    .option("header","true")
                    .option("inferSchema","true")
                    .option("sep",",")
                    .load(file_path)
              )

# PS. See above another way to write long commands in many lines, with the help of ( )
```

In [25]:
```
# First, let us check the schema and the initial lines of the dataset.
# We should take this step...

flight_data.
flight_data.
```

```
root
 |-- FL_DATE: date (nullable = true)
 |-- OP_CARRIER: string (nullable = true)
 |-- OP_CARRIER_FL_NUM: integer (nullable = true)
 |-- ORIGIN: string (nullable = true)
 |-- DEST: string (nullable = true)
 |-- CRS_DEP_TIME: integer (nullable = true)
 |-- DEP_TIME: double (nullable = true)
 |-- DEP_DELAY: double (nullable = true)
 |-- TAXI_OUT: double (nullable = true)
 |-- WHEELS_OFF: double (nullable = true)
 |-- WHEELS_ON: double (nullable = true)
 |-- TAXI_IN: double (nullable = true)
 |-- CRS_ARR_TIME: integer (nullable = true)
 |-- ARR_TIME: double (nullable = true)
 |-- ARR_DELAY: double (nullable = true)
 |-- CANCELLED: double (nullable = true)
 |-- CANCELLATION_CODE: string (nullable = true)
 |-- DIVERTED: double (nullable = true)
 |-- CRS_ELAPSED_TIME: double (nullable = true)
 |-- ACTUAL_ELAPSED_TIME: double (nullable = true)
 |-- AIR_TIME: double (nullable = true)
 |-- DISTANCE: double (nullable = true)
 |-- CARRIER_DELAY: double (nullable = true)
 |-- WEATHER_DELAY: double (nullable = true)
 |-- NAS_DELAY: double (nullable = true)
 |-- SECURITY_DELAY: double (nullable = true)
 |-- LATE_AIRCRAFT_DELAY: double (nullable = true)
 |-- Unnamed: 27: string (nullable = true)

+----------+----------+-----------------+------+----+------------+--------+---------+--------+----------+--------
|   FL_DATE|OP_CARRIER|OP_CARRIER_FL_NUM|ORIGIN|DEST|CRS_DEP_TIME|DEP_TIME|DEP_DELAY|TAXI_OUT|WHEELS_OFF|WHEELS_O
+----------+----------+-----------------+------+----+------------+--------+---------+--------+----------+--------
|2017-01-01|        AA|                1|   JFK| LAX|         800|   831.0|     31.0|    25.0|     856.0|   1143.
|2017-01-01|        AA|                2|   LAX| JFK|         900|   934.0|     34.0|    34.0|    1008.0|   1757.
|2017-01-01|        AA|                4|   LAX| JFK|        1130|  1221.0|     51.0|    20.0|    1241.0|   2025.
|2017-01-01|        AA|                5|   DFW| HNL|        1135|  1252.0|     77.0|    19.0|    1311.0|   1744.
|2017-01-01|        AA|                6|   OGG| DFW|        1855|  1855.0|      0.0|    16.0|    1911.0|    631.
+----------+----------+-----------------+------+----+------------+--------+---------+--------+----------+--------
only showing top 5 rows
```

In [26]:
```
# Also, we can have a variable with the columns

cols = flight_data.columns
```

In [27]:
```
# Just a detail: to figure out how, for example, sorting by FLIGHTS would work

flight_data.sort("OP_CARRIER").explain() # check the Spark physical plan
```

```
== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- Sort [OP_CARRIER#554 ASC NULLS FIRST], true, 0
   +- Exchange rangepartitioning(OP_CARRIER#554 ASC NULLS FIRST, 6), ENSURE_REQUIREMENTS, [plan_id=312]
      +- FileScan csv [FL_DATE#553,OP_CARRIER#554,OP_CARRIER_FL_NUM#555,ORIGIN#556,DEST#557,CRS_DEP_TIME#558,DEP_
```

Before moving on, a note about reading data from a csv file:

Above, we have inferred the schema from the first line of the csv file. And by the way reading is a transformation not an action.

But we could have set the schema programatically and then read the data from the file accordingly. When schema is inferred from a huge file this may take some time. So in those circunstances we may decide to set the schema programmatically.

In [28]:
```
# Specify the DataFrame with data of interest
# We can select a subset or drop a subset!

cols_interest = ["FL_DATE", "OP_CARRIER", "OP_CARRIER_FL_NUM", "ORIGIN", "DEST",
     "CRS_DEP_TIME", "DEP_TIME", "DEP_DELAY",
     "CRS_ARR_TIME", "ARR_TIME", "ARR_DELAY",
     "CRS_ELAPSED_TIME", "ACTUAL_ELAPSED_TIME",
     "AIR_TIME", "DISTANCE"]

flight_data = flight_data.select(cols_interest)
flight_data.columns
```

Out[28]:
```
['FL_DATE',
 'OP_CARRIER',
 'OP_CARRIER_FL_NUM',
 'ORIGIN',
 'DEST',
 'CRS_DEP_TIME',
 'DEP_TIME',
 'DEP_DELAY',
 'CRS_ARR_TIME',
 'ARR_TIME',
 'ARR_DELAY',
 'CRS_ELAPSED_TIME',
 'ACTUAL_ELAPSED_TIME',
 'AIR_TIME',
 'DISTANCE']
```

In [29]:
```
# Add four more columns with, respectively in relation to the flight date,
# the year, the month, the day of the week and the week of the year

flight_data = ( flight_data
                .withColumn("FL_DATE_YEAR", F.year("FL_DATE"))
                .withColumn("FL_DATE_MONTH",
                .withColumn("FL_DATE_DAY_OF_WEEK",
                .withColumn("FL_DATE_WEEK_OF_YEAR",
                )
```

Note: See functions related to Timestamp fields in

https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.to_timestamp.html

https://spark.apache.org/docs/latest/sql-ref-datetime-pattern.html

https://spark.apache.org/docs/latest/sql-ref-functions-builtin.html#date-and-timestamp-functions

In [30]:
```
# Check the 1st record of data now

flight_data.show(
```

```
-RECORD 0-------------------------
 FL_DATE               | 2017-01-01
 OP_CARRIER            | AA
 OP_CARRIER_FL_NUM     | 1
 ORIGIN                | JFK
 DEST                  | LAX
 CRS_DEP_TIME          | 800
 DEP_TIME              | 831.0
 DEP_DELAY             | 31.0
 CRS_ARR_TIME          | 1142
 ARR_TIME              | 1209.0
 ARR_DELAY             | 27.0
 CRS_ELAPSED_TIME      | 402.0
 ACTUAL_ELAPSED_TIME   | 398.0
 AIR_TIME              | 347.0
 DISTANCE              | 2475.0
 FL_DATE_YEAR          | 2017
 FL_DATE_MONTH         | 1
 FL_DATE_DAY_OF_WEEK   | 1
 FL_DATE_WEEK_OF_YEAR | 52
only showing top 1 row
```

Questions to be answered

How many flights are recorded in the dataset?

In [ ]:
```
flight_data.
```

What are the average of departure delay (DEP_DELAY) and the average of arrival delay (ARR_DELAY)?

In [32]:
```
flight_data.select(F.avg("DEP_DELAY"), F.avg("ARR_DELAY")).show()
```

```
[Stage 28:>                                                         (0 + 8) / 8]
```

```
+-----------------+---------------+
|   avg(DEP_DELAY)| avg(ARR_DELAY)|
+-----------------+---------------+
|9.725734044679225|4.3263569087054|
+-----------------+---------------+
```

Which are the top 3 busiests airports regarding departures?

In [ ]:
```
flight_data.
```

Give an airport such that no other airport have more arrivals?

In [ ]:
```
flight_data.
```

Which airline holds the highest number of departures in an airport, and which airport is?

In [ ]:
```
flight_data.
```

In [36]:
```
# Finally, we may stop the SparkSession if we want

spark.stop()
```

Answers regarding the questions above

How many flights are recorded in the dataset?

answer: 5674621

What are the average of departure delay (DEP_DELAY) and the average of arrival delay (ARR_DELAY)?

answer:

Average departure delay	Average arrival delay
9.725734044679225	4.3263569087054

Which are the top 3 busiests airpoirts regarding departures?

answer:

Origin	count
ATL	364655
ORD	266460
DEN	223165

Give an airport such that no other airport have more arrivals?

answer: ATL (364596 flights)

Which airline holds the highest number of departures in an airport, and which airport is?

answer: carrier=DL, airport=ATL (242493 flights)

Additional exercise

Reading a directory of data at once

In the case of having many files in one directory and all sharing the same data schema, we can read all files at once into one data frame. Of course, in practical terms, the resulting data frame may be too big. (Try using Pandas!)

With data regarding US flights delays that have been provided, read the complete dataset for many years with a single read command.

Hint: Try by indicating the directory where all files are located, instead of making reference to a specific filename.

See https://spark.apache.org/docs/latest/sql-data-sources-generic-options.html#path-global-filter

References

- Learning Spark - Lightning-Fast Data Analytics, 2nd Ed. J. Damji, B. Wenig, T. Das, and D. Lee. O'Reilly, 2020
- https://spark.apache.org/docs/latest
- https://docs.python.org/3/
- https://www.kaggle.com/datasets/yuanyuwendymu/airline-delay-and-cancellation-data-2009-2018?select=2018.csv