An Introduction to Apache, PySpark and Dataframe Transformations

A Comprehensive Guide to Master Big Data Analysis

Victor Roman Follow

Jun 12, 2019 · 10 min read ★

Introduction: The Big Data Problem

Apache arises as a new engine and programming model for data analytics. It's origin goes back to 2009, and the main reasons why it has gained so much importance in the past recent years are due to changes in enconomic factors that underline computer applications and hardware.

Historically, the power of computers only grew with time. Each year, new processors were able to perform operations faster and the applications that run on top of them automatically got faster.

All of this changed in 2005, when the limits in heat disipation caused the switch from making individual processors faster, to start exploring the parallelization of CPU cores. This meant that applications and the code that run them must be changed too. All of this is what layed out the ground of new models like Apache Spark.

In addition, the cost of sensors and storing units only had decreased on the last years. Nowadays is completely unexpensive to collect and store vast amounts of information.

There is so much data available, that the way to process it and analyze it, must change radically too, by making large parallel computations on clusters of cumputers. These clusters enable the synergic combination of those computers' power, simultaneously, and make much easier tackling expensive computational tasks like data processing.

And this is where Apache Spark comes into play.

What is Apache Spark

As found on the great book: Spark — The Definitive Guide:

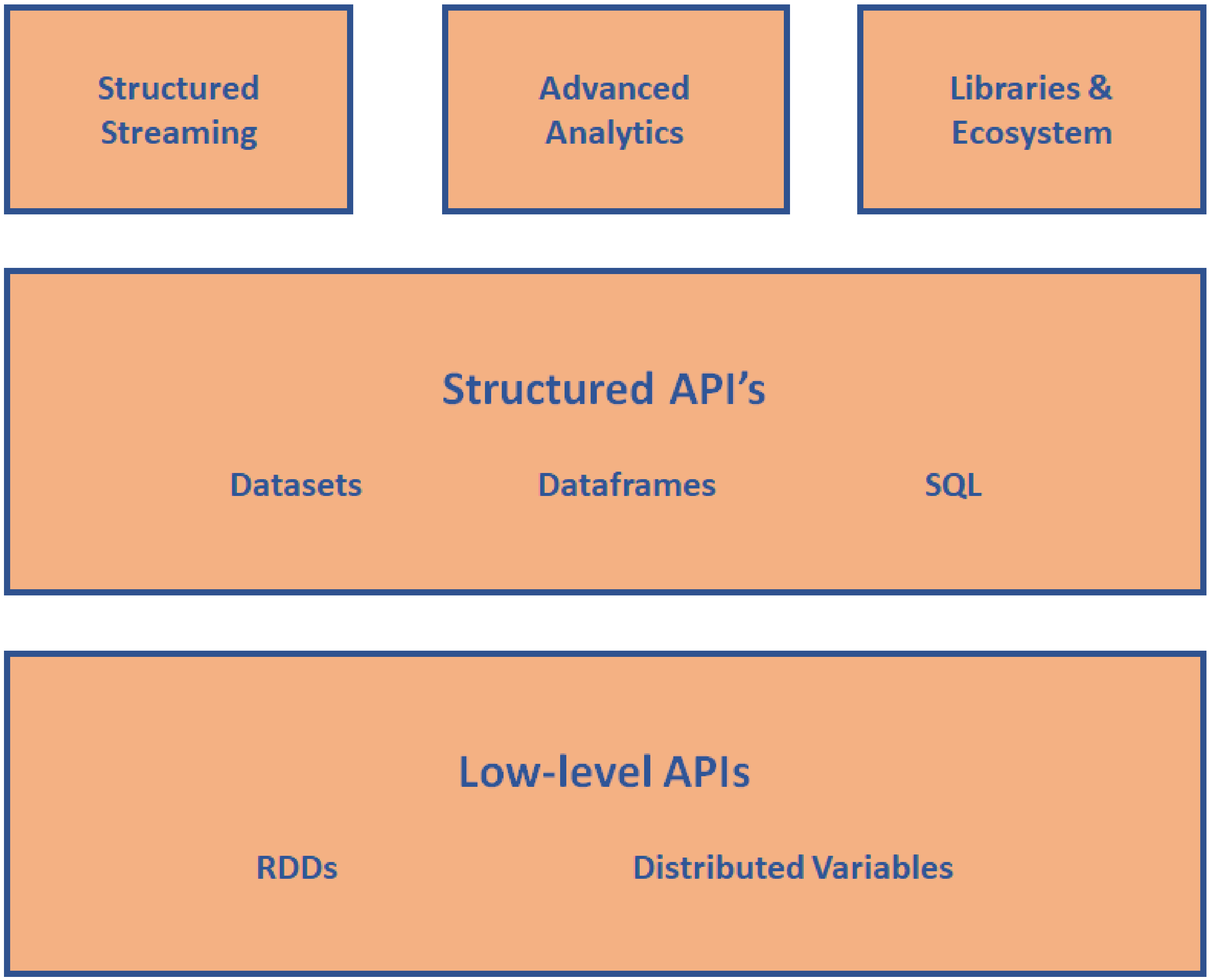
> "Apache Spark is a unified computing engine and a set of libraries for parallel data procesing on clusters of computers"

Nowadays, Apache Spark is the most popular open source engine to Big Data processing. And the main reasons are:

- It supports programming languages as widely used as: Python, Scala, Java and R.
- It supports SQL tasks.
- It enables data streaming.

- It has libraries for Machine Learning and Deep Learning.
- It can be run in a single machine or in a cluster of computers.

The following is an sketch that illustrates the different libraries available in the Spark ecosystem.



How to Set Up and Run Apache Spark

Throughout these series of articles, we will focus on Apache Spark Python's library, PySpark. As stated before, Spark can be run both locally and in a cluster of computers. There are several ways to configure our machines to run Spark locally, but are out of the scope of these articles.

One of the simplest and fastest ways to work with PsyPark and unlock its inmense processing power, is with the free website Databricks, concretely by using its Community Edition.

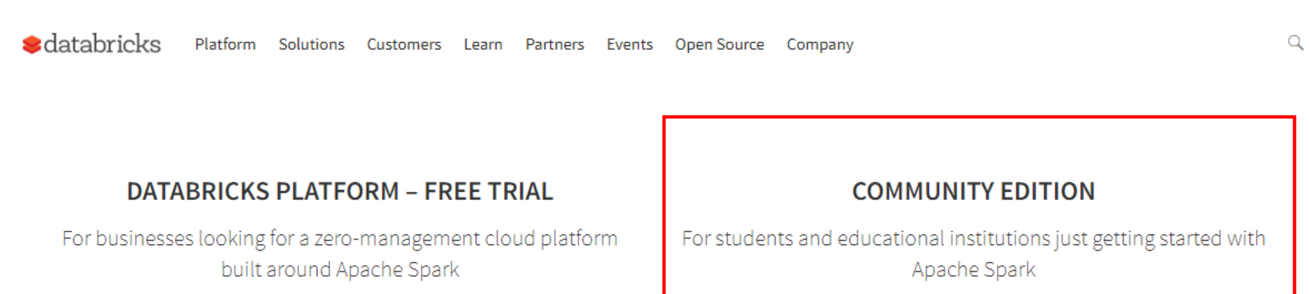
To get started we shoud simply go to:

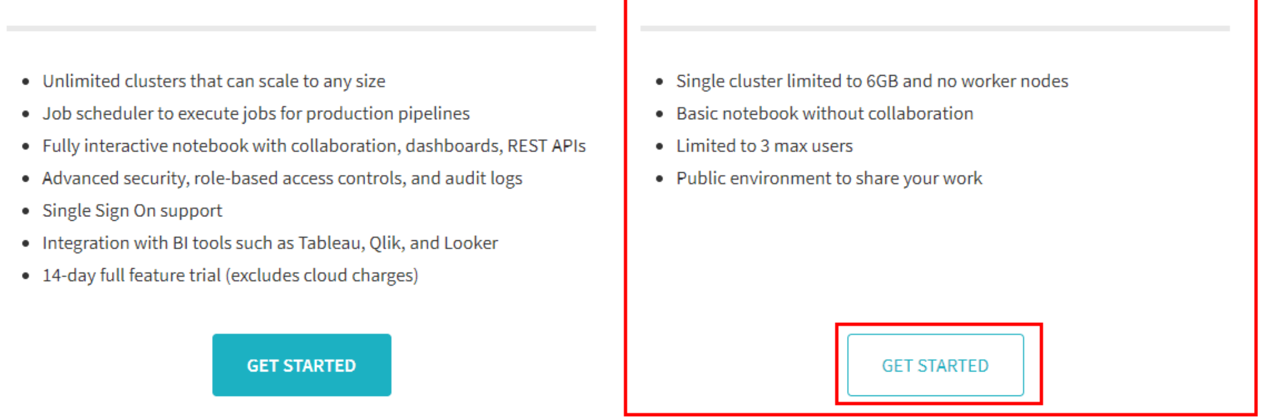
Try Databricks

Unlimited clusters that can scale to any size Job scheduler to execute jobs for production pipelines Fully interactive...

databricks.com

And select its Community Edition:

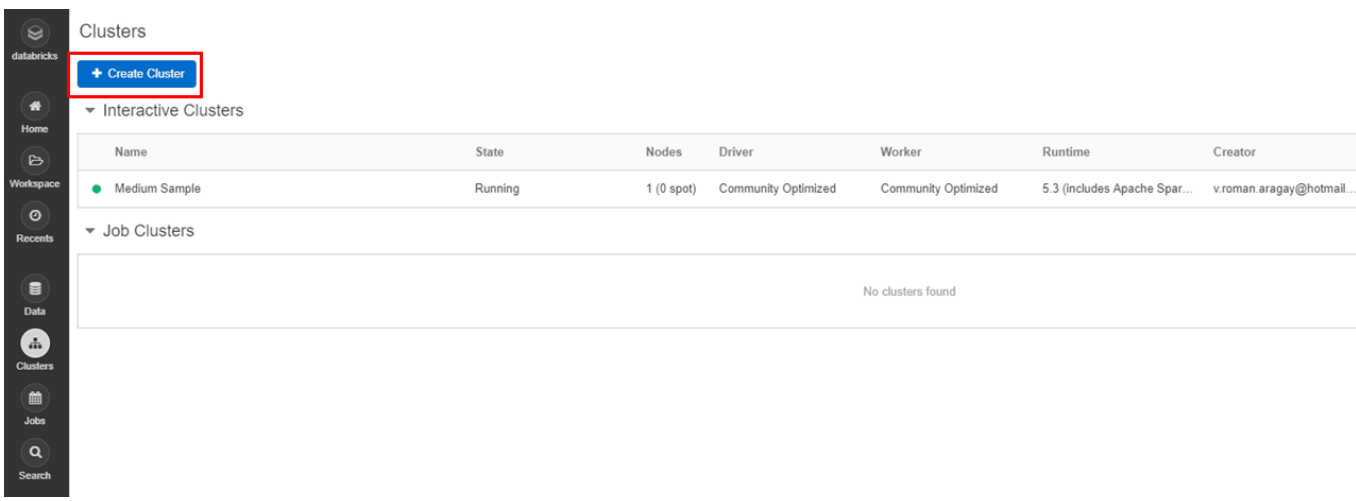




Then, we must create and account.

Running A Temporal Cluster

Once we have created an account, to be able to start working, we shoud create a temporary cluster.



As it is a free version, these clusters have a default of 6 Gb of RAM and can be run for 6 hours each. In order to develop industrial projects or work with Data Pipelines, it is suggested to use the premiun platform.

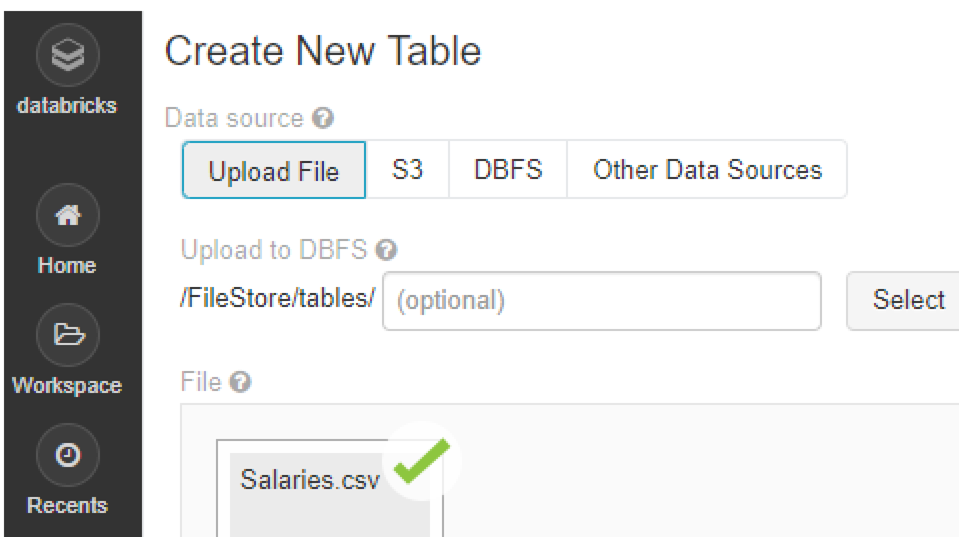
But for the aim of these tutorials, the community edition will be more than enough.

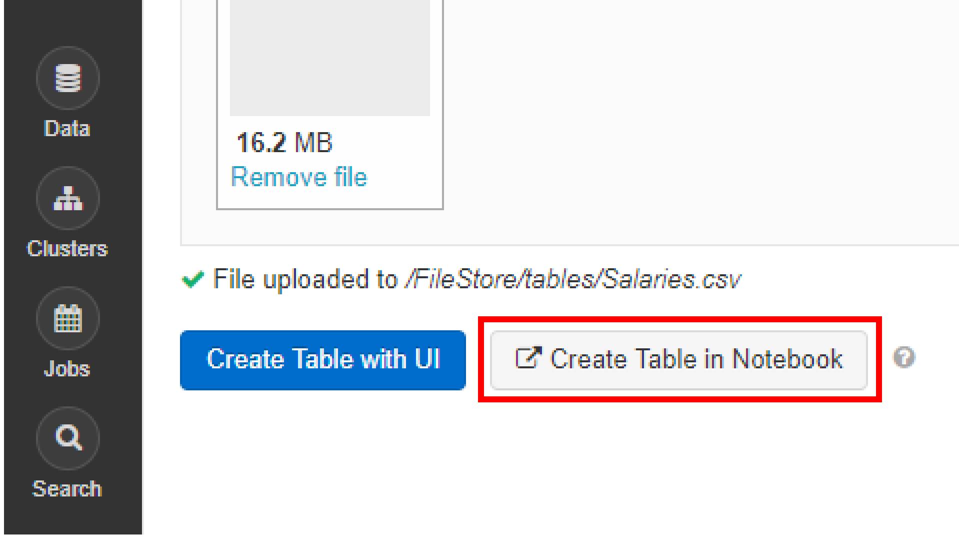
Adding Data

In order to add data to work with:

- Click on the data tab
- Then add data

You can work both with available data uploaded by other users or with data uploaded from your computer.



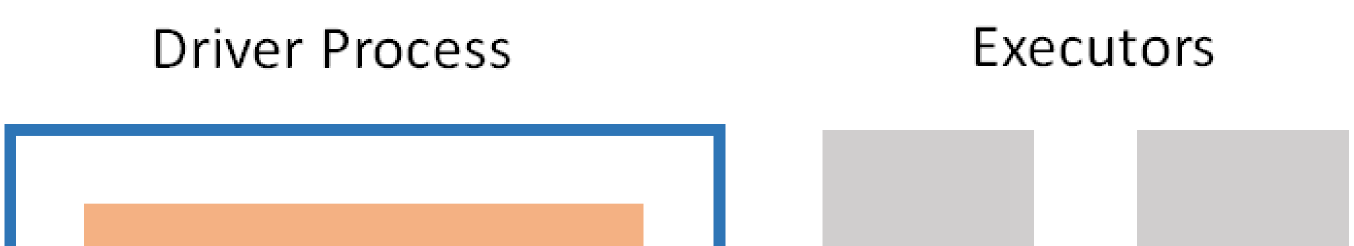


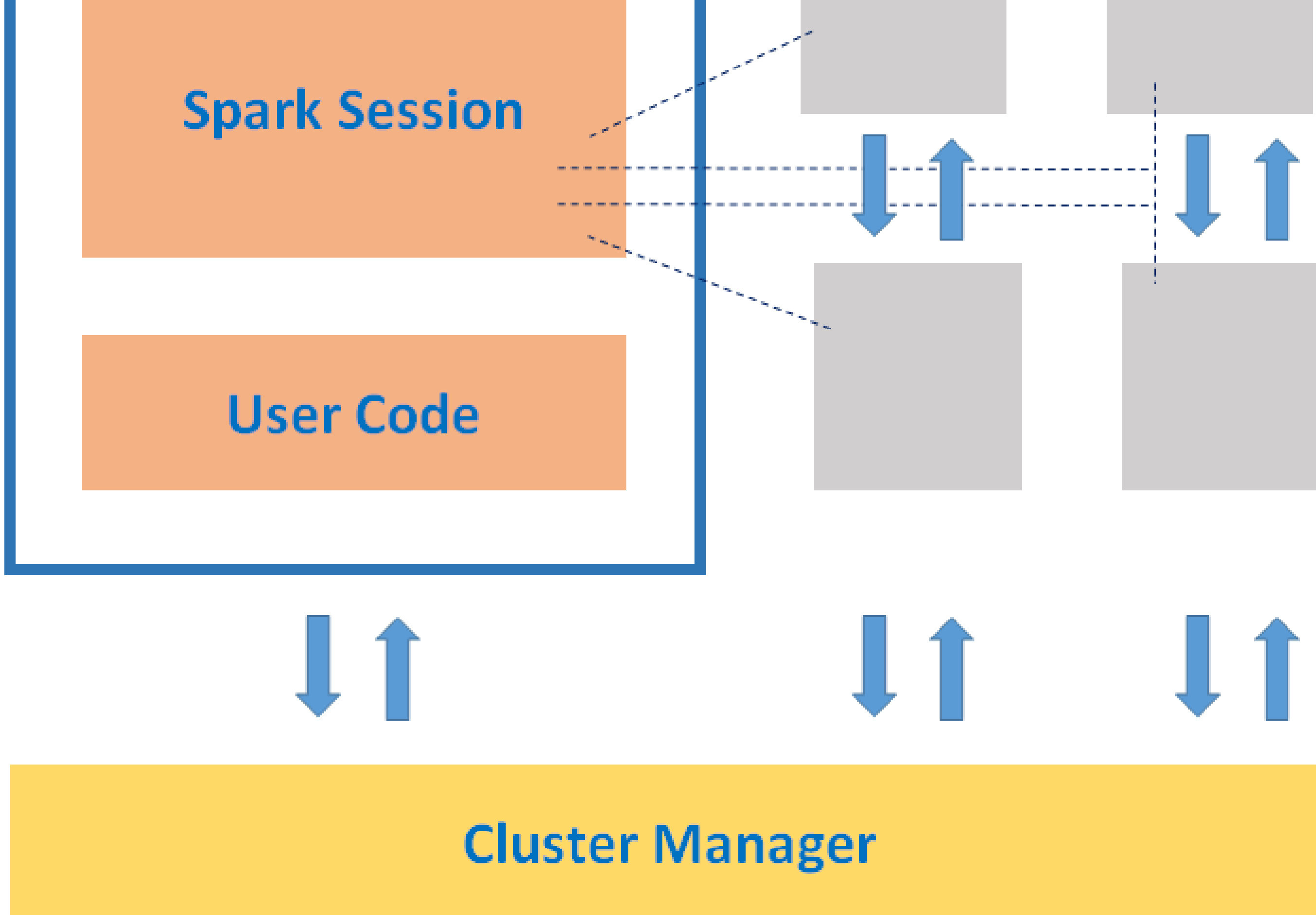
Once, done we can create a Table in a Notebook and we are all set up!

Pyspark Applications & Partitions

To understand how Apache Spark works we should talk about the core components of a Spark Application: The Driver, the Executors and the Cluster Manager.

The following is a very illustrative sketch of a Spark Application Architecture:





Driver

The driver is located in a node of the cluster of computers and performs three main tasks:

1. Holds information about the Spark Application
2. Responds to a input, for example a user's programm
3. Analyzes, distributes and schedules the tasks to be done by the executers.

Executors

The executors are the ones that actually perform the work assigned by the driver. They do two things:

1. Executing the code assigned to them.
2. Reporting the state of the computation to the driver.

Cluster Manager

The cluster manager is the responsible for:

1. Controlling the physical computers
2. Distributing resources to Spark Applications

There can be several Spark Applications running on the same cluster, at the same time, and all of them will be managed by the Cluster Manager.

PySpark Dataframes

Apache Spark works with several data abstractions, each with an specific interface to work with. The most common abstractions are:

- Datasets
- Dataframes

- SQL Tables
- Resilient Distributed Datasets

Throughout these series we will focus on the most common unit to represent and store data in Apache Spark, Dataframes.

Dataframes are data tables with rows and columns, the closest analogy to understand them are spreadsheets with labeled columns.

One important feature of Dataframes is their schema. A Dataframe's schema is a list with its columns names and the type of data that each column stores.

Other relevant attribute of Dataframes is that they are not located in one simple computer, in fact they can be splitted through hundreds of machines. This is due to optimize the processing of the information and when data is too large to fit a single machine.

Apache Partitions

As stated before, the executors perform the work assigned by the driver, and they do it in a parallel fashion, in order to be able to do this, Spark split data into different partitions.

These partitions are collections of rows located in a single computer within a cluster. When we talk about Dataframe's partitions we are talking about how the data is distributed across all the machines on our cluster.

Most of the time we will not specify explicitly how the partitions will be done in our clusters, but with our code we will transmit high-level transformations of the data and Spark will realize by itself which is the optimal way to perform these partitions. Always looking for obtaining the maximum processing efficiency.

Low level APIs to perform these operations are out of the scope of these series.

Dataframes Transformations

First of all, we have to understand that transformations are modifications that we specify to do to our dataframes.

These transformations are specified in a high-level fashion and will not be executed until we explicitely call for an action to be made.

This way of working is called lazy evaluation, and the aim is to improve efficiency. When we call for transformations to be made, Spark will desing a plan to perform optimally these tasks, and will not execute it until the very last minute when we call an action (like .show() or .collect())

Apple Stock Price

Now, we will explore some of the most common actions and transformations. We are going to work with Apple stock price's data, from 2010 to 2016. We will perform some exploratory data analysis, data transformations, deal with missing values and perform grouping and aggregating.

Import Dataframe

To initialize and display a datframe, the code will be the following:

```
# File location and type
file_location = "/FileStore/tables/appl_stock.csv"
file_type = "csv"

# CSV options
infer_schema = "true"
first_row_is_header = "true"
delimiter = ","

# The applied options are for CSV files. For other file types, these
will be ignored.
df = spark.read.format(file_type) \
  .option("inferSchema", infer_schema) \
  .option("header", first_row_is_header) \
  .option("sep", delimiter) \
  .load(file_location)

# Display Dataframe
display(df)
```

(3) Spark Jobs

df: pyspark.sql.dataframe.DataFrame = [Date: timestamp, Open: double ... 5 more fields]

Date	Open	High	Low	Close	Volume	Adj Close
2010-01-04T00:00:00.000+0000	213.429998	214.499996	212.38000099999996	214.009998	123432400	27.727039
2010-01-05T00:00:00.000+0000	214.599998	215.589994	213.249994	214.379993	150476200	27.774976000000002
2010-01-06T00:00:00.000+0000	214.379993	215.23	210.750004	210.969995	138040000	27.333178000000004
2010-01-07T00:00:00.000+0000	211.75	212.000006	209.050005	210.58	119282800	27.28265
2010-01-08T00:00:00.000+0000	210.299994	212.000006	209.06000500000002	211.98000499999998	111902700	27.464034
2010-01-11T00:00:00.000+0000	212.79999700000002	213.000002	208.450005	210.11000299999998	115557400	27.221758
2010-01-12T00:00:00.000+0000	209.18999499999998	209.76999500000002	206.419998	207.720001	148614900	26.91211
2010-01-13T00:00:00.000+0000	207.870005	210.92999500000002	204.099998	210.650002	151473000	27.29172
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Showing the first 1000 rows.

Get Dataframe's Schema

The schema of a dataframe is the description of the structure of the data, it is a collection of StructField objects and provides information about the type of the data in a dataframe.

To diplay the Dataframe's Schema is as simple as:

```
# Display Dataframe's Schema
df.printSchema()
```

```
root
 |-- Date: timestamp (nullable = true)
 |-- Open: double (nullable = true)
 |-- High: double (nullable = true)
 |-- Low: double (nullable = true)
 |-- Close: double (nullable = true)
 |-- Volume: integer (nullable = true)
 |-- Adj Close: double (nullable = true)
```

Perform Filtering and Transformations

To filter our data, to get only those rows that have a closing price smaller that $500, we could run the following line of code:

```
# Filter data usign pyspark
df.filter(" Close < 500").show())
```

Date	Open	High	Low	Close	Volume	Adj Close
2010-01-04 00:00:00	213.429998	214.499996	212.38000099999996	214.009998	123432400	27.727039
2010-01-05 00:00:00	214.599998	215.589994	213.249994	214.379993	150476200	27.774976000000002
2010-01-06 00:00:00	214.379993	215.23	210.750004	210.969995	138040000	27.333178000000004
2010-01-07 00:00:00	211.75	212.000006	209.050005	210.58	119282800	27.28265
2010-01-08 00:00:00	210.299994	212.000006	209.06000500000002	211.98000499999998	111902700	27.464034
2010-01-11 00:00:00	212.79999700000002	213.000002	208.450005	210.11000299999998	115557400	27.221758
2010-01-12 00:00:00	209.18999499999998	209.76999500000002	206.419998	207.720001	148614900	26.91211
2010-01-13 00:00:00	207.870005	210.92999500000002	204.099998	210.650002	151473000	27.29172
2010-01-14 00:00:00	210.11000299999998	210.45999700000002	209.020004	209.43	108223500	27.133657
2010-01-15 00:00:00	210.92999500000002	211.59999700000003	205.869999	205.93	148516900	26.680197999999997
2010-01-19 00:00:00	208.330002	215.18999900000003	207.240004	215.039995	182501900	27.860484999999997
2010-01-20 00:00:00	214.910006	215.549994	209.500002	211.73	153038200	27.431644
2010-01-21 00:00:00	212.079994	213.30999599999998	207.210003	208.069996	152038600	26.957455
2010-01-22 00:00:00	206.78000600000001	207.499996	197.16	197.75	220441900	25.620401
2010-01-25 00:00:00	202.51000200000001	204.699999	200.190002	203.070002	266424900	26.309658000000002
2010-01-26 00:00:00	205.95000100000001	213.710005	202.580004	205.940001	466777500	26.681494
2010-01-27 00:00:00	206.849995	210.58	199.530001	207.880005	430642100	26.932840000000002
2010-01-28 00:00:00	204.930004	205.500004	198.699995	199.289995	293375600	25.819922000000002
2010-01-29 00:00:00	201.079996	202.199995	190.250002	192.060003	311488100	24.883208
2010-02-01 00:00:00	192.36999699999998	196.0	191.29999899999999	194.729998	187469100	25.229131

only showing top 20 rows

We can also filter to only obtain certain columns:

```
# Filter data by columns
df.filter("Close < 500").select(['Open', 'Close']).show()
```

```
+------------------+------------------+
|              Open|             Close|
+------------------+------------------+
|        213.429998|        214.009998|
|        214.599998|        214.379993|
|        214.379993|        210.969995|
|            211.75|            210.58|
|        210.299994|211.98000499999998|
|212.79999700000002|210.11000299999998|
|209.18999499999998|        207.720001|
|        207.870005|        210.650002|
|210.11000299999998|            209.43|
|210.92999500000002|            205.93|
|        208.330002|        215.039995|
|        214.910006|            211.73|
|        212.079994|        208.069996|
|206.78000600000001|            197.75|
|202.51000200000001|        203.070002|
|205.95000100000001|        205.940001|
|        206.849995|        207.880005|
|        204.930004|        199.289995|
|        201.079996|        192.060003|
|192.36999699999998|        194.729998|
+------------------+------------------+
only showing top 20 rows
```

To filter by one column and showing other, we will use the .select() mehtod.

```
# Filter by one column and show other
df.filter(df['Close'] < 500).select('Volume').show()
```

```
+---------+
|   Volume|
+---------+
|123432400|
```

```
|150476200|
|138040000|
|119282800|
|111902700|
|115557400|
|148614900|
|151473000|
|108223500|
|148516900|
|182501900|
|153038200|
|152038600|
|220441900|
|266424900|
|466777500|
|430642100|
|293375600|
|311488100|
|187469100|
+---------+
only showing top 20 rows
```

To filter by multiple conditions:

```
# Filter by multiple conditions: closing price < $200 and opening price > $200
df.filter( (df['Close'] < 200) & (df['Open'] > 200) ).show()
```

```
+-------------------+------------------+----------+----------+----------+---------+------------------+
|               Date|              Open|      High|       Low|     Close|   Volume|         Adj Close|
+-------------------+------------------+----------+----------+----------+---------+------------------+
|2010-01-22 00:00:00|206.78000600000001|207.499996|    197.16|    197.75|220441900|         25.620401|
|2010-01-28 00:00:00|        204.930004|205.500004|198.699995|199.289995|293375600|25.819922000000002|
|2010-01-29 00:00:00|        201.079996|202.199995|190.250002|192.060003|311488100|         24.883208|
+-------------------+------------------+----------+----------+----------+---------+------------------+
```

Obtain a Statistic Summary of the Data

Similarly to other libraries likePandas, we can obtain a statistic summary of the Dataframe by simply running the .describe() method.

```
# Display Statistic Summary
df.describe().show()
```

summary	Open	High	Low	Close	Volume	Adj Close
count	1762	1762	1762	1762	1762	1762
mean	313.0763111589103	315.9112880164581	309.8282405079457	312.9270656379113	9.422577587968218E7	75.00174115607275
stddev	185.29946803981522	186.89817686485767	183.38391664371008	185.1471036170943	6.020518776592709E7	28.57492972179906
min	90.0	90.699997	89.470001	90.279999	11475900	24.881912
max	702.409988	705.070023	699.569977	702.100021	470249500	127.96609099999999

Add and Rename Columns

To add a new column to the dataframe, we will use the .withColumn() method as follows.

```
# Display Dataframe with new column
df.withColumn('Doubled Adj Close', df['Adj Close']*2).select('Adj Close', 'Doubled Adj Close').show()
```

```
+------------------+------------------+
|         Adj Close| Doubled Adj Close|
```

```
+------------------+------------------+
|         27.727039|         55.454078|
|27.774976000000002|55.549952000000005|
|27.333178000000004| 54.66635600000001|
|          27.28265|           54.5653|
|         27.464034|         54.928068|
|         27.221758|         54.443516|
|          26.91211|          53.82422|
|          27.29172|          54.58344|
|         27.133657|         54.267314|
|26.680197999999997|53.360395999999994|
|27.860484999999997|55.720969999999994|
|         27.431644|         54.863288|
|         26.957455|          53.91491|
|         25.620401|         51.240802|
|26.309658000000002|52.619316000000005|
|         26.681494|         53.362988|
|26.932840000000002|53.865680000000005|
|25.819922000000002|51.639844000000004|
|         24.883208|         49.766416|
|         25.229131|         50.458262|
+------------------+------------------+
only showing top 20 rows
```

To rename an existing column, we will use the .withColumnRenamed() method.

```
# Display Dataframe with renamed column
df.withColumnRenamed('Adj Close', 'Adjusted Close Price').show()
```

```
+-------------------+------------------+------------------+------------------+------------------+---------+--------------------+
|               Date|              Open|              High|               Low|             Close|   Volume|Adjusted Close Price|
+-------------------+------------------+------------------+------------------+------------------+---------+--------------------+
|2010-01-04 00:00:00|        213.429998|        214.499996|212.38000099999996|        214.009998|123432400|           27.727039|
|2010-01-05 00:00:00|        214.599998|        215.589994|        213.249994|        214.379993|150476200|  27.774976000000002|
|2010-01-06 00:00:00|        214.379993|            215.23|        210.750004|        210.969995|138040000|  27.333178000000004|
|2010-01-07 00:00:00|            211.75|        212.000006|        209.050005|            210.58|119282800|            27.28265|
```

2010-01-08 00:00:00	210.299994	212.000006	209.06000500000002	211.98000499999998	111902700	27.464034
2010-01-11 00:00:00	212.79999700000002	213.000002	208.450005	210.11000299999998	115557400	27.221758
2010-01-12 00:00:00	209.18999499999998	209.76999500000002	206.419998	207.720001	148614900	26.91211
2010-01-13 00:00:00	207.870005	210.92999500000002	204.099998	210.650002	151473000	27.29172
2010-01-14 00:00:00	210.11000299999998	210.45999700000002	209.020004	209.43	108223500	27.133657
2010-01-15 00:00:00	210.92999500000002	211.59999700000003	205.869999	205.93	148516900	26.680197999999997
2010-01-19 00:00:00	208.330002	215.18999900000003	207.240004	215.039995	182501900	27.860484999999997
2010-01-20 00:00:00	214.910006	215.549994	209.500002	211.73	153038200	27.431644
2010-01-21 00:00:00	212.079994	213.30999599999998	207.210003	208.069996	152038600	26.957455
2010-01-22 00:00:00	206.78000600000001	207.499996	197.16	197.75	220441900	25.620401
2010-01-25 00:00:00	202.51000200000001	204.699999	200.190002	203.070002	266424900	26.309658000000002
2010-01-26 00:00:00	205.95000100000001	213.710005	202.580004	205.940001	466777500	26.681494
2010-01-27 00:00:00	206.849995	210.58	199.530001	207.880005	430642100	26.932840000000002
2010-01-28 00:00:00	204.930004	205.500004	198.699995	199.289995	293375600	25.819922000000002
2010-01-29 00:00:00	201.079996	202.199995	190.250002	192.060003	311488100	24.883208
2010-02-01 00:00:00	192.36999699999998	196.0	191.29999899999999	194.729998	187469100	25.229131

Grouping and Aggregating Data

Now, we will perform some gruping and aggretion of our data, in order to obtain meaningful insights. But first, we should import some libraries

```
# Import relevant libraries
from pyspark.sql.functions import
dayofmonth,hour,dayofyear,weekofyear,month,year,format_number,date_fo
rmat,mean, date_format, datediff, to_date, lit
```

Now, let us create a new column, with the year of each row:

```
# To know th average closing price per year
new_df = df.withColumn('Year', year(df['Date']))
new_df.show()
```

Date	Open	High	Low	Close	Volume	Adj Close	Year
2010-01-04 00:00:00	213.429998	214.499996	212.38000099999996	214.009998	123432400	27.727039	2010
2010-01-05 00:00:00	214.599998	215.589994	213.249994	214.379993	150476200	27.774976000000002	2010
2010-01-06 00:00:00	214.379993	215.23	210.750004	210.969995	138040000	27.333178000000004	2010
2010-01-07 00:00:00	211.75	212.000006	209.050005	210.58	119282800	27.28265	2010
2010-01-08 00:00:00	210.299994	212.000006	209.06000500000002	211.98000499999998	111902700	27.464034	2010
2010-01-11 00:00:00	212.79999700000002	213.000002	208.450005	210.11000299999998	115557400	27.221758	2010
2010-01-12 00:00:00	209.18999499999998	209.76999500000002	206.419998	207.720001	148614900	26.91211	2010
2010-01-13 00:00:00	207.870005	210.92999500000002	204.099998	210.650002	151473000	27.29172	2010
2010-01-14 00:00:00	210.11000299999998	210.45999700000002	209.020004	209.43	108223500	27.133657	2010
2010-01-15 00:00:00	210.92999500000002	211.59999700000003	205.869999	205.93	148516900	26.680197999999997	2010
2010-01-19 00:00:00	208.330002	215.18999900000003	207.240004	215.039995	182501900	27.860484999999997	2010
2010-01-20 00:00:00	214.910006	215.549994	209.500002	211.73	153038200	27.431644	2010
2010-01-21 00:00:00	212.079994	213.30999599999998	207.210003	208.069996	152038600	26.957455	2010
2010-01-22 00:00:00	206.78000600000001	207.499996	197.16	197.75	220441900	25.620401	2010
2010-01-25 00:00:00	202.51000200000001	204.699999	200.190002	203.070002	266424900	26.309658000000002	2010
2010-01-26 00:00:00	205.95000100000001	213.710005	202.580004	205.940001	466777500	26.681494	2010
2010-01-27 00:00:00	206.849995	210.58	199.530001	207.880005	430642100	26.932840000000002	2010
2010-01-28 00:00:00	204.930004	205.500004	198.699995	199.289995	293375600	25.819922000000002	2010
2010-01-29 00:00:00	201.079996	202.199995	190.250002	192.060003	311488100	24.883208	2010
2010-02-01 00:00:00	192.36999699999998	196.0	191.29999899999999	194.729998	187469100	25.229131	2010

Now, lets group by this recently created 'Year' column and aggreagate by the maximum, minimum and average prices of each year to obtain meaningful insights of the status and evolution of the price.

```
# Group and aggregate data
new_df.groupBy('Year').agg(f.max('Close').alias('Max Close'),
f.min('Close').alias('Min Close'), f.mean('Close').alias('Average
Close')).orderBy('Year').show()
```

Year	Max Close	Min Close	Average Close
2010	325.470013	192.050003	259.8424600000002
2011	422.23999800000007	315.320007	364.00432532142867
2012	702.100021	411.23	576.0497195640002
2013	570.090004	390.530006	472.6348802857143

```
|2014|         647.349983| 90.279999| 295.4023416507935|
|2015|              133.0|103.120003|120.03999980555547|
|2016|             118.25| 90.339996|104.60400786904763|
+----+-------------------+----------+------------------+
```

We have achieved our goal! However, we still have some very difficult data to read. In fact we have way more decimals than we need.

Taking into account that we are working with prices of hundreds of dollars, more than two decimals do not provide us with relevant information.

So let's take advantage and learn to format the results to show us the number of decimals we want.

Formating Our Data

To format our data we will use the format_number() function as follows:

```
# Import relevant functions
from pyspark.sql.functions import forman_number, col

# Select the appropiate columns to format
cols = ['Max Close', 'Min Close', 'Average Close']

# Format the columns
formatted_df = new_df.select('Year', *[format_number(col(col_name),
2).name(col_name) for col_name in cols])
```

```
+----+---------+---------+-------------+
|Year|Max Close|Min Close|Average Close|
+----+---------+---------+-------------+
|2010|   325.47|   192.05|       259.84|
|2011|   422.24|   315.32|       364.00|
|2012|   702.10|   411.23|       576.05|
|2013|   570.09|   390.53|       472.63|
|2014|   647.35|    90.28|       295.40|
|2015|   133.00|   103.12|       120.04|
|2016|   118.25|    90.34|       104.60|
+----+---------+---------+-------------+
```

User Defined Functions

Let's learn now how to apply functions defined by us to our dataframes. We will use it in this example to get a column with the month of the year in which each row was recorded.

```
# Import relevant functions
from pyspark.sql.functions import date_format, datediff, to_date,
lit, UserDefinedFunction, month
from pyspark.sql.types import StringType
from pyspark.sql import functions as F

# Create month list
month_lst = ['January', 'Feburary', 'March', 'April', 'May', 'June',
'July', 'August', 'September', 'October', 'November', 'December']

# Define the function
udf = UserDefinedFunction(lambda x: month_lst[int(x%12) - 1],
StringType())
```

```
# Add column to df with the number of the month of the year
df = df.withColumn('moy_number', month(df.Date))

# Apply function and generate a column with the name of the month of
the year
df = df.withColumn('moy_name', udf("moy_number"))
```

Date	Open	High	Low	Close	Volume	Adj Close	moy_number	moy_name
2010-01-04 00:00:00	213.429998	214.499996	212.38000099999996	214.009998	123432400	27.727039	1	January
2010-01-05 00:00:00	214.599998	215.589994	213.249994	214.379993	150476200	27.774976000000002	1	January
2010-01-06 00:00:00	214.379993	215.23	210.750004	210.969995	138040000	27.333178000000004	1	January
2010-01-07 00:00:00	211.75	212.000006	209.050005	210.58	119282800	27.28265	1	January
2010-01-08 00:00:00	210.299994	212.000006	209.06000500000002	211.98000499999998	111902700	27.464034	1	January
2010-01-11 00:00:00	212.79999700000002	213.000002	208.450005	210.11000299999998	115557400	27.221758	1	January
2010-01-12 00:00:00	209.18999499999998	209.76999500000002	206.419998	207.720001	148614900	26.91211	1	January
2010-01-13 00:00:00	207.870005	210.92999500000002	204.099998	210.650002	151473000	27.29172	1	January
2010-01-14 00:00:00	210.11000299999998	210.45999700000002	209.020004	209.43	108223500	27.133657	1	January
2010-01-15 00:00:00	210.92999500000002	211.59999700000003	205.869999	205.93	148516900	26.680197999999997	1	January
2010-01-19 00:00:00	208.330002	215.18999900000003	207.240004	215.039995	182501900	27.860484999999997	1	January
2010-01-20 00:00:00	214.910006	215.549994	209.500002	211.73	153038200	27.431644	1	January
2010-01-21 00:00:00	212.079994	213.30999599999998	207.210003	208.069996	152038600	26.957455	1	January
2010-01-22 00:00:00	206.78000600000001	207.499996	197.16	197.75	220441900	25.620401	1	January
2010-01-25 00:00:00	202.51000200000001	204.699999	200.190002	203.070002	266424900	26.309658000000002	1	January
2010-01-26 00:00:00	205.95000100000001	213.710005	202.580004	205.940001	466777500	26.681494	1	January
2010-01-27 00:00:00	206.849995	210.58	199.530001	207.880005	430642100	26.932840000000002	1	January
2010-01-28 00:00:00	204.930004	205.500004	198.699995	199.289995	293375600	25.819922000000002	1	January
2010-01-29 00:00:00	201.079996	202.199995	190.250002	192.060003	311488100	24.883208	1	January
2010-02-01 00:00:00	192.36999699999998	196.0	191.29999899999999	194.729998	187469100	25.229131	2	Feburary

Success!

Conclusion

Throughout this article we have covered:

- The basis of Apache Spark
- We have gained an intuition of why it is important and how it operates

- Perform analysis operations with PySpark and Dataframes

On the next articles we will learn how to apply Machine Learning in PySpark and apply this knowledge to some projects. Stay tuned!

Big Data Data Science Pyspark Data Analysis Spark

Discover Medium

Welcome to a place where words matter. On Medium, smart voices and original ideas take center stage - with no ads in sight. Watch

Make Medium yours

Follow all the topics you care about, and we'll deliver the best stories for you to your homepage and inbox. Explore

Become a member

Get unlimited access to the best stories on Medium — and support writers while you're at it. Just $5/month. Upgrade

About Help Legal