**Creating a DataFrame**

In the following, I would like to create a fake DataFrame which contains fake bought and sold stock assets.

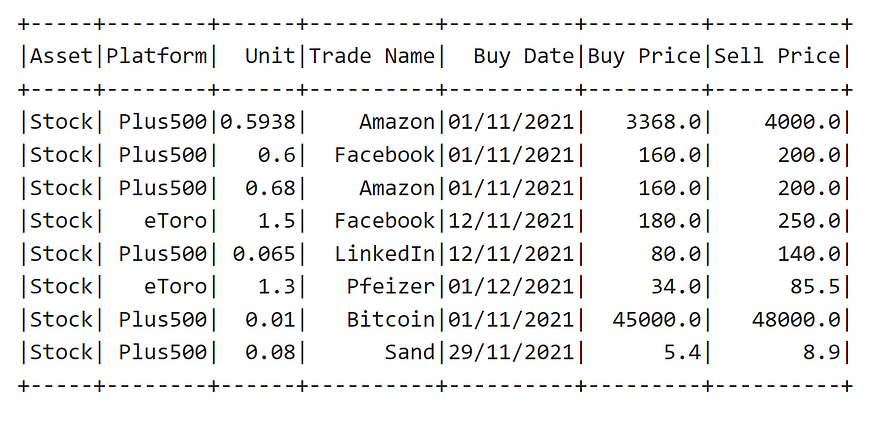
**First approach of a creating a DataFrame— Creating DataFrame using Tuple and .toDF() function**

The first approach for creating a data frame in Spark using Scala syntax is to use the *spark.implicits.\_.*

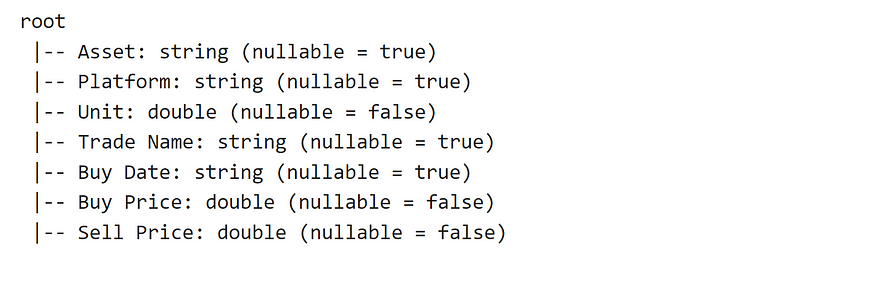
In this approach, each row of the data frame corresponds to a tuple in which we bring the name of the columns in the .toDF() function. Let us create a DataFrame with a few rows using the following code snippet:

import spark.implicits.\_  
   
val data = Seq(  
 ("Stock", "Plus500", 0.5938, "Amazon", "01/11/2021", 3368.000, 4000.0),  
 ("Stock", "Plus500", 0.6, "Facebook", "01/11/2021", 160.0, 200.0),  
 ("Stock", "Plus500", 0.68, "Amazon", "01/11/2021", 160.0, 200.0),  
 ("Stock", "eToro", 1.5, "Facebook", "12/11/2021", 180.0, 250.0),  
 ("Stock", "Plus500", 0.065, "LinkedIn", "12/11/2021", 80.0, 140.0),  
 ("Stock", "eToro", 1.3, "Pfeizer", "01/12/2021", 34.0, 85.5),  
 ("Stock", "Plus500", 0.01, "Bitcoin", "01/11/2021", 45000.0, 48000.0),  
 ("Stock", "Plus500", 0.08, "Sand", "29/11/2021", 5.4, 8.9)  
 )  
  
val firstApproachDF = data.toDF("Asset", "Platform", "Unit", "Trade Name", "Buy Date", "Buy Price", "Sell Price")  
  
firstApproachDF.show()  
  
firstApproachDF.printSchema

The output of the above code snippet is as follows:



Created dataframe with 8 rows and 7 columns



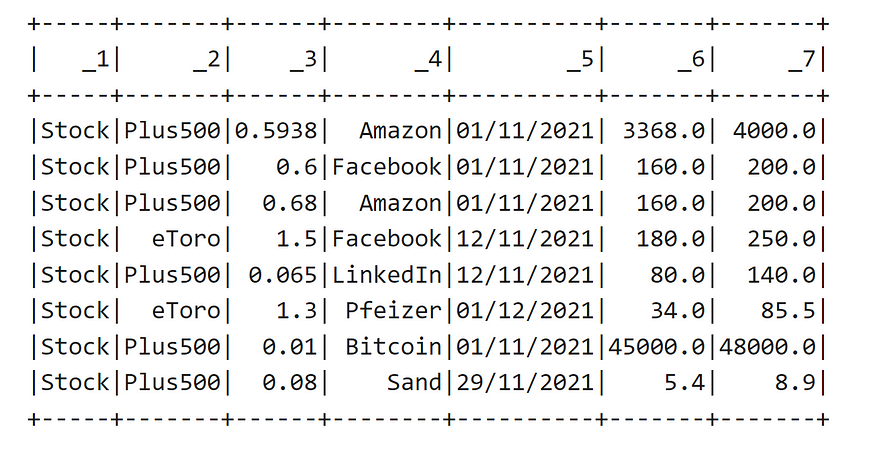
The schema of the created dataframe

**Second approach of creating dataframe — Using createDataFrame() function**

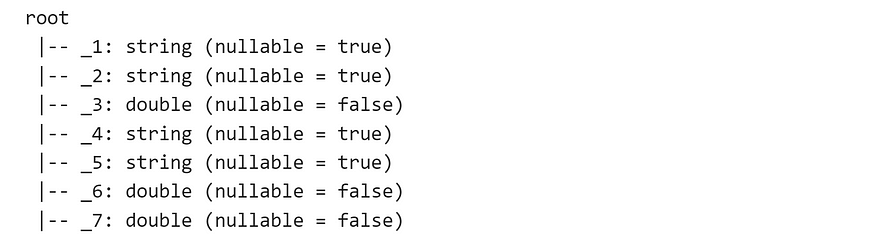
In the second approach, we use spark.createDataFrame() to create a DataFrame. The data still contains the sequence of tuples in Scala like the following:

val data = Seq(  
 ("Stock", "Plus500", 0.5938, "Amazon", "01/11/2021", 3368.000, 4000.0),  
 ("Stock", "Plus500", 0.6, "Facebook", "01/11/2021", 160.0, 200.0),  
 ("Stock", "Plus500", 0.68, "Amazon", "01/11/2021", 160.0, 200.0),  
 ("Stock", "eToro", 1.5, "Facebook", "12/11/2021", 180.0, 250.0),  
 ("Stock", "Plus500", 0.065, "LinkedIn", "12/11/2021", 80.0, 140.0),  
 ("Stock", "eToro", 1.3, "Pfeizer", "01/12/2021", 34.0, 85.5),  
 ("Stock", "Plus500", 0.01, "Bitcoin", "01/11/2021", 45000.0, 48000.0),  
 ("Stock", "Plus500", 0.08, "Sand", "29/11/2021", 5.4, 8.9)  
 )  
   
 val secondApproachDF = spark.createDataFrame(data)  
   
   
 secondApproachDF.show()  
 secondApproachDF.printSchema

However, there is a difference in the output, and you can see that we do not have the column names anymore as indicated in the following output:



Created dataframe using CreateDataFrame()



output — schema without column names

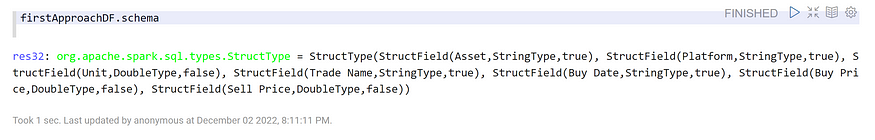
In the above case, it might be needed to rename the columns or define a schema.

**Schema and columns of a DataFrame**

It is always possible to use the following methods to see the schema and columns of a DataFrame in Spark. Let us consider the firstAppachDF defined in previous sub section as follows:

**Schema of DataFrame**

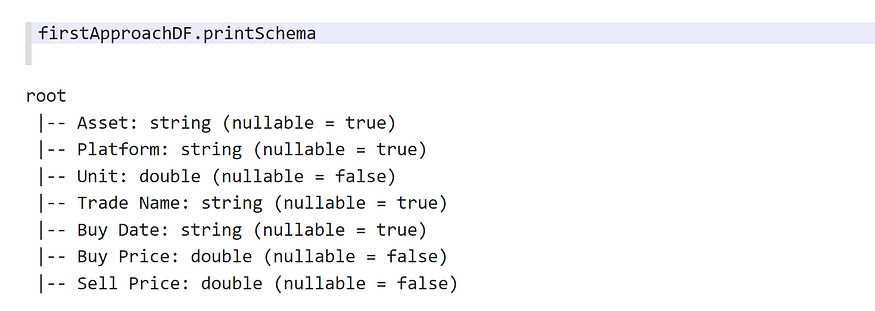
firstApproachDF.schema



out put — .schema to see the schema in terms of data types

**Printing schema of the DataFrame**

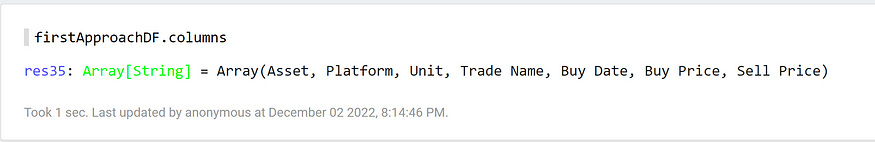
firstApproach.printSchema



ouptut — printing the schema

**Columns of a DataFrame**

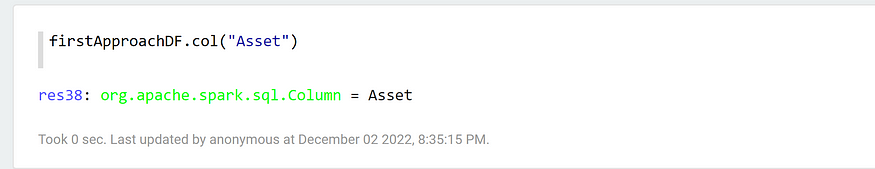
firstApproachDF.columns



output — firstApproachDF columns

**A single column**

firstApproachDF.col("Asset")

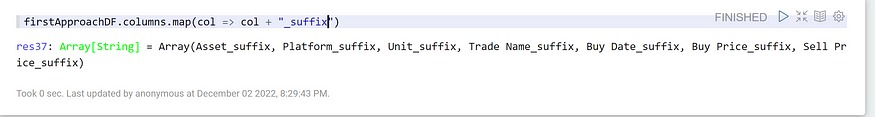


output — Accessing a single column

**Iterating over columns of a DataFrame using Map function and appending values**

Sometimes, it is required to go through every column of a DataFrame and append a suffix. It is possible to achieve that using a map() function like below:

firstApproachDF.columns.map(col => col + "\_suffix")



output — \_suffix is visible in column names

From now on let us use **DF**instead of **firstApproachDF**.

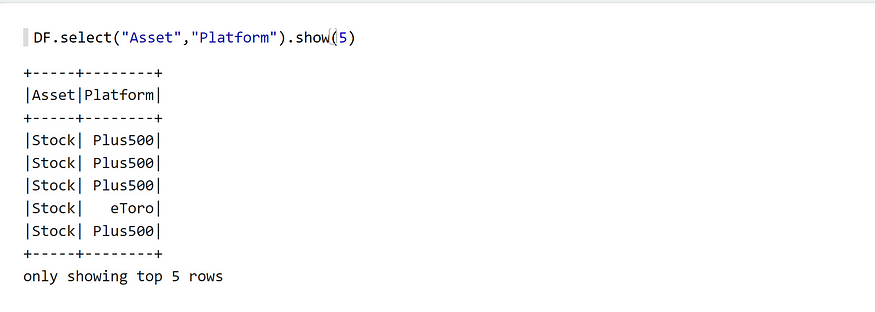
**Various approaches for selecting columns of a DataFrame**

There are various ways to select columns of a DataFrame. Selecting the columns of a DataFrame is important due to two aspects:

1. You might see various approaches in code so it is better to see these formats.
2. You may need to add some sort of dynamics to your code in order to expand a DataFrame.

**Approach 1. using select and column names**

DF.select("Asset","Platform"l).show(5)

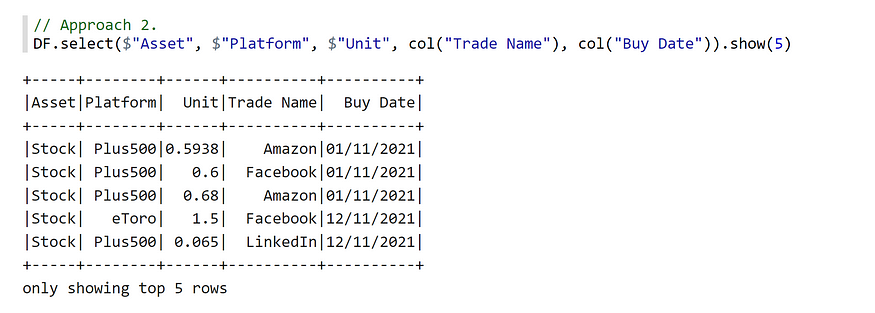


showing 5 rows of Asset and Platform Columns

**Approach 2 and 3. using select with $ sign or col( )**

I brought the second and third approaches as it is possible to use them together like the following:

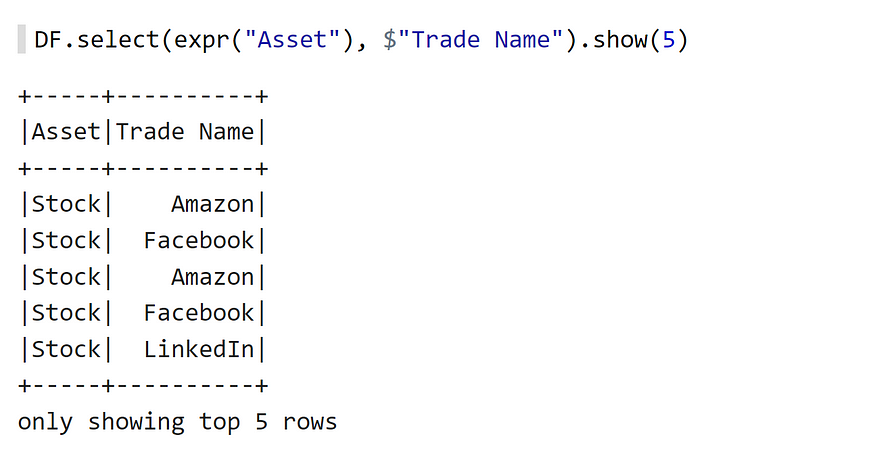
DF.select($"Asset", $"Platform", $"Unit", col("Trade Name"),   
col("Buy Date")).show(5)



output — using $ and col to select columns

**Approach 4. using expressions and spark implicits**

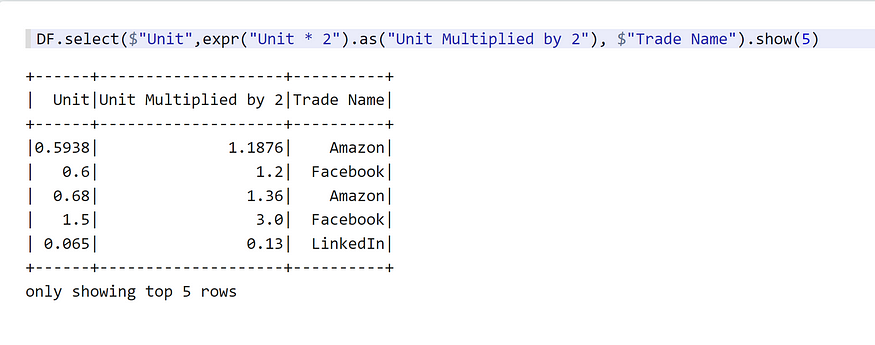
DF.select(expr("Asset"), $"Trade Name").show(5)



output — selecting using expr

It is also possible to apply calculations using *expr*. Let us consider the following:

DF.select($"Unit",expr("Unit \* 2").as("Unit Multiplied by 2"),   
$"Trade Name").show(5)

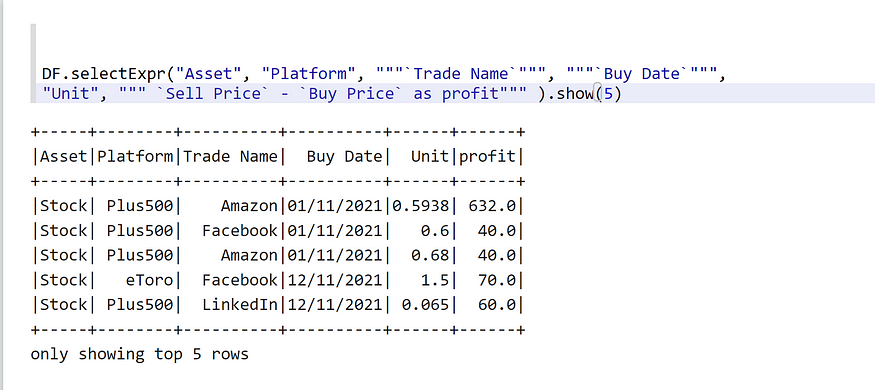


Output — Multiplying Unit by 2 using expr

**Approach 5. using selectExpr**

Let us consider the following example of calculating profit from the difference of *Buy Price* and *Sell Price*using*selectExpr* function. Please note that we should use `` for columns with spaces:

DF.selectExpr("Asset", "Platform", """`Trade Name`""", """`Buy Date`""",   
"Unit", """ `Sell Price` - `Buy Price` as profit""" ).show(5)

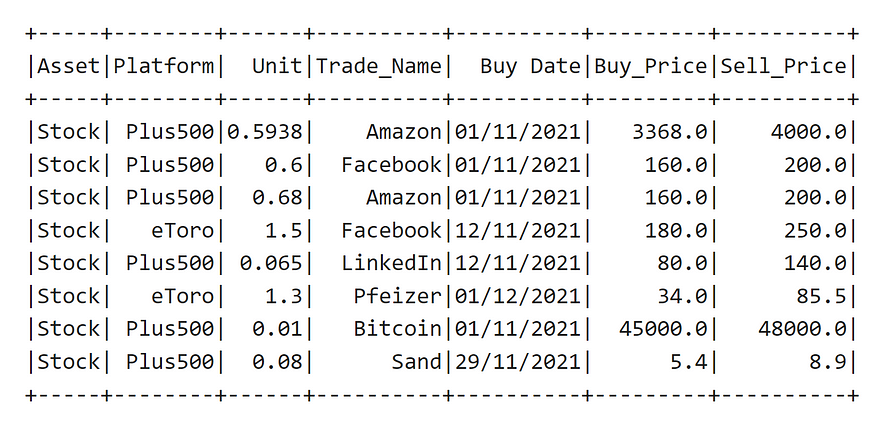


Using selectExpr

**Renaming a column and creating a new column**

It is possible to rename the column name using *.withColumnRenamed()* function, so let us have a look at it by adding underscore (\_) to the column names with spaces:

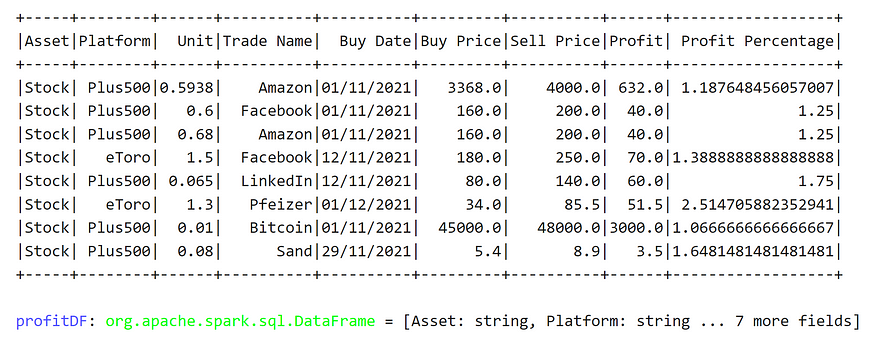
val renamedColumns = DF.withColumnRenamed("Trade Name","Trade\_Name")  
.withColumnRenamed("Buy Price","Buy\_Price")  
.withColumnRenamed("Sell Price","Sell\_Price")  
  
renamedColumns.show()



output — renamed Columns

For creating a new column, we can use the function *.withColumn()*in which we can specify the column name as well. In the following, we create two columns named “*Profit*” and “*Profit Percentage*”

val profitDF = DF.withColumn("Profit", col("Sell Price") - col("Buy Price"))  
.withColumn("Profit Percentage", col("Sell Price") / col("Buy Price"))  
  
profitDF.show()



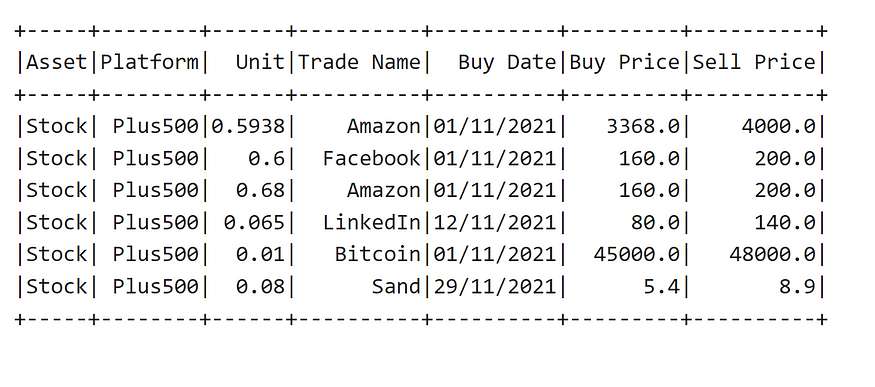
output — Adding two new columns named Profit and Profit Percentage

**Filtering rows in Spark**

Oftentimes, it is required to filter rows of DataFrame based on certain criteria. For doing so, it is possible to use *filter*function in which in the following, I review some example of filter functions:

**Example 1**

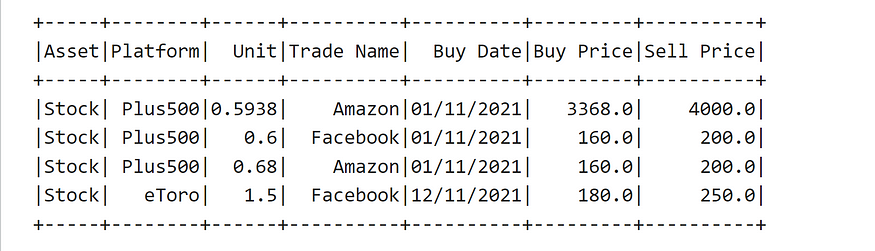
DF.filter(col("Platform") === "Plus500").show()



Keeping only the Plus500 rows

**Example 2. using OR operator (||)**

val orExampleDF = DF.filter(col("Trade Name") === "Amazon"   
|| col("Trade Name") ==="Facebook")  
  
orExampleDF.show()



Filtering Facebook or Amazon

**Example 3. using AND operator (&&)**

Keeping only *Facebook*for the *Trade Name* and *Plus500*as the platform:

val andExampleDF = DF.filter(col("Trade Name") === "Facebook"   
&& col("Platform") ==="Plus500")  
  
andExampleDF.show()

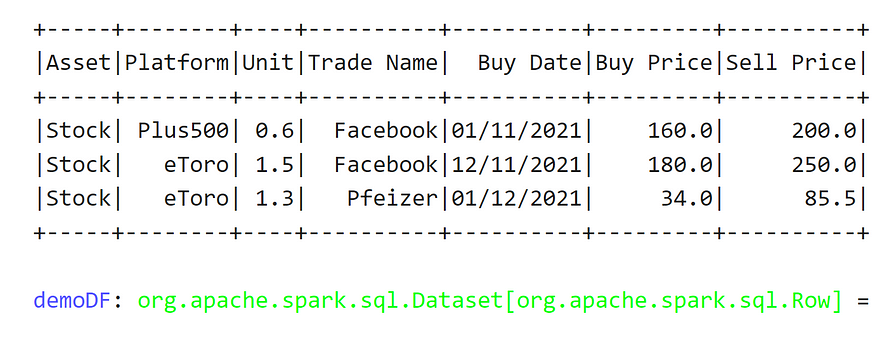


output — And Example

**Example 4. Not equal and greater than (>)**

Let us select all trades with more than 0.5 unit in which the trade name is not Amazon:

val demoDF = DF.filter(col("Unit") >= 0.5 && col("Trade Name") =!= "Amazon")  
  
demoDF.show()



output — Example of not equal and greater than (>)

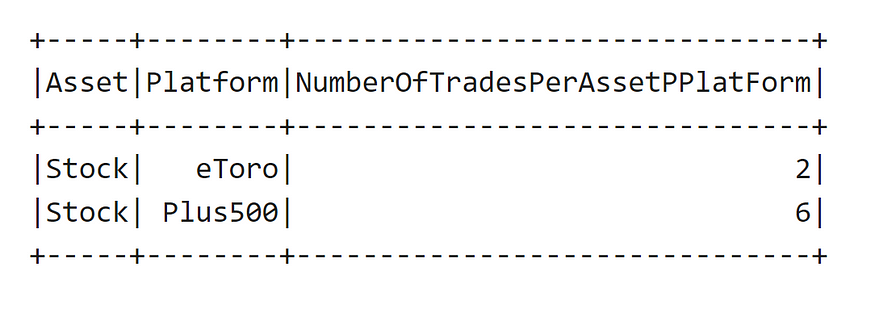
**Spark grouping and aggregation functions**

It is very much common to calculate several statistics over groups of data. Spark has a very rich support for such calculations. Considering the DF from previous sections, let us calculate several statistics:

**Example of count()**

Here, we group by per *Asset*and *Platform*and count the number of trades.

// Calculating number of trades per platform  
  
val numberOfTradesPerPlatform = DF.groupBy("Asset", "Platform")  
.agg(count("\*").as("NumberOfTradesPerAssetPPlatForm"))  
  
numberOfTradesPerPlatform.show()

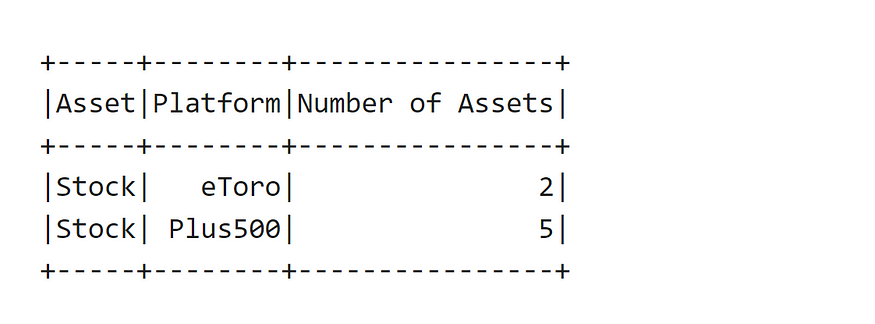


output

**Example of countDistinct()**

Here, we group by per Asset and Platform and count the distinct values of traded items:

// Count distinct of number of traded items  
val numberOfAssets = DF.groupBy("Asset","Platform")  
.agg(countDistinct("Trade Name").as("Number of Assets"))  
  
numberOfAssets.show()

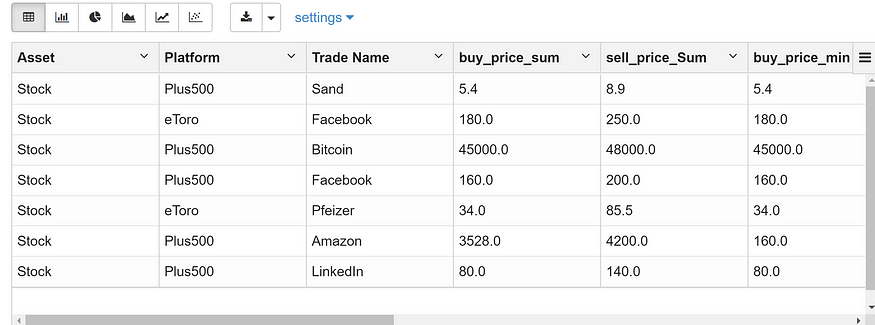


output — countDistinct

**Example of Sum, AVG, Median, Min, Max**

In the following, we calculate *sum, avg, min, max and P50* of buy and sell prices over over Asset, Platform and Trade Name:

val agg\_df = DF.groupBy("Asset", "Platform", "Trade Name").agg(  
   
   
 sum("Buy Price").as("buy\_price\_sum"),  
 sum("Sell Price").as("sell\_price\_Sum"),  
   
 min("Buy Price").as("buy\_price\_min"),  
 min("Sell Price").as("sell\_price\_min"),  
   
 max("Buy Price").as("buy\_price\_max"),  
 max("Sell Price").as("sell\_price\_max"),  
   
 avg("Buy Price").as("buy\_price\_avg"),  
 avg("Sell Price").as("sell\_price\_avg"),  
   
 expr("percentile(`Sell Price`, 0.5)").as("sell\_price\_P50"),  
 expr("percentile(`Buy Price`, 0.5)").as("buy\_price\_P50")  
 )  
   
 z.show(agg\_df)



output — only some of the columns are shown

**Spark dynamic groupBy and aggregation**

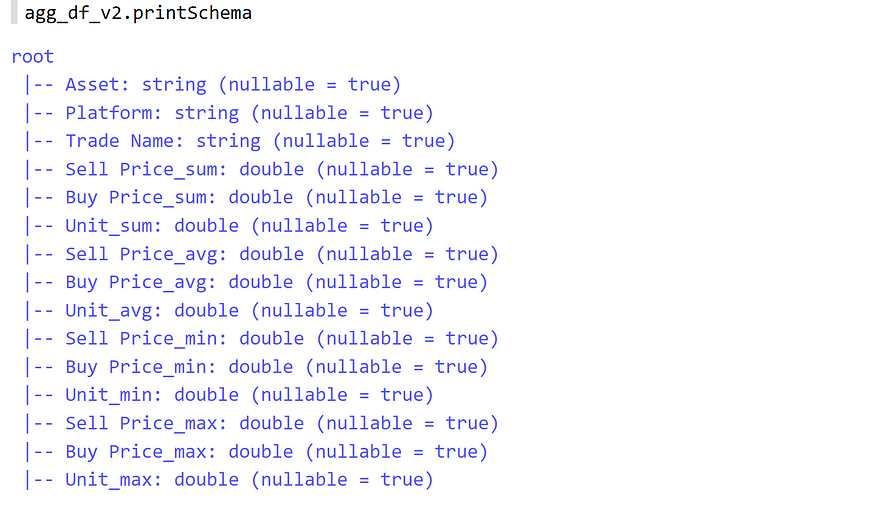
On many occasions, you would like to perform aggregations on a lot of columns e.g., between 10 to 100 columns, and you would like to apply different aggregation functions, so it would not be practical to write all the code so it would be more efficient to use a bit of syntactic sugar using the map function. [In another tutorial](https://medium.com/@clever.tech.memes/application-of-map-function-in-dynamic-spark-groupby-and-aggregations-cbe67f4ca753), I have explained in detail the Spark dynamic groupBy aggregation. I try to explain with an example but if you need further explanation, please refer to the link above.

Let us now perform the same groupBy and aggregation from the exact previous section as follows:

val intended\_columns = List("Sell Price","Buy Price", "Unit")  
  
val sumExpr = intended\_columns.map(col => sum(col).as(col + "\_sum"))  
val avgExpr = intended\_columns.map(col => avg(col).as(col + "\_avg"))  
val minExpr = intended\_columns.map(col => min(col).as(col + "\_min"))  
val maxExpr = intended\_columns.map(col => max(col).as(col + "\_max"))  
  
val aggExpression = sumExpr ++ avgExpr ++ minExpr ++ maxExpr  
  
val agg\_df\_v2 = DF.groupBy("Asset", "Platform", "Trade Name")  
.agg(aggExpression.head, aggExpression.tail: \_\*)  
  
  
agg\_df\_v2.printSchema

We define a list of columns that are interested in performing aggregations i.e., *Sell Price, Buy Price, and Unit*. Now, we define the expressions. An example is *sumExpr*in which it iterates over all the columns in the *intended\_columns*, sums the column value, and renames it. Afterward, we create a final *aggExpression*which is the concatenation of all the other expressions. We use *aggExpression.head and aggExpression.tail in the agg() function together with “:\_\*”* to consider all the expressions.

The output is very huge so I cannot print all the columns, but here is the schema:



Schema of the created df

So, you can imagine if we have a lot of columns, how much flexibility this approach can give us. We can also add the same flexibility to the list of column names in the *groupBy*expression.

**Spark Join**

You can perform joins in two ways in which is brought by an example:

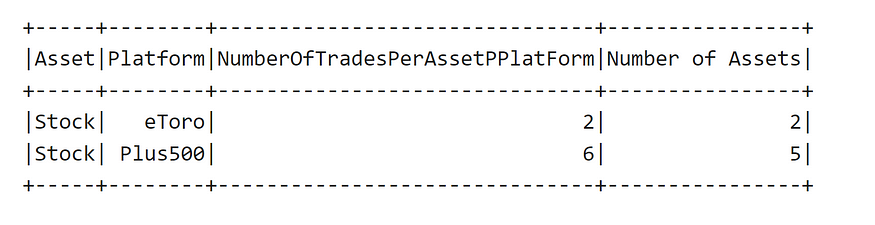
We saw these two DataFrames in previous sections:

val numberOfTradesPerPlatform = DF.groupBy("Asset", "Platform")  
.agg(count("\*").as("NumberOfTradesPerAssetPPlatForm"))  
  
  
  
val numberOfAssets = DF.groupBy("Asset","Platform")  
.agg(countDistinct("Trade Name").as("Number of Assets"))

Now, let us have a look at how to perform the join:

**First approach for join using Seq()**

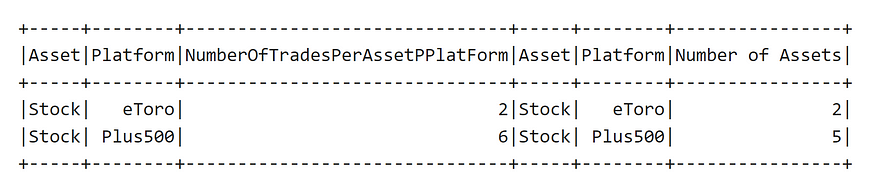
// First approach for join  
  
val joinedDF = numberOfTradesPerPlatform  
.join(numberOfAssets, Seq("Asset","Platform"))  
  
joinedDF.show()



join using Seq()

**Second Approach for join**

// Second approach for join  
  
val joinedDF2 = numberOfTradesPerPlatform.join(numberOfAssets,   
numberOfTradesPerPlatform.col("Asset") === numberOfAssets.col("Asset")   
&& numberOfTradesPerPlatform.col("Platform") === numberOfAssets.col("Platform"))  
joinedDF2.show()

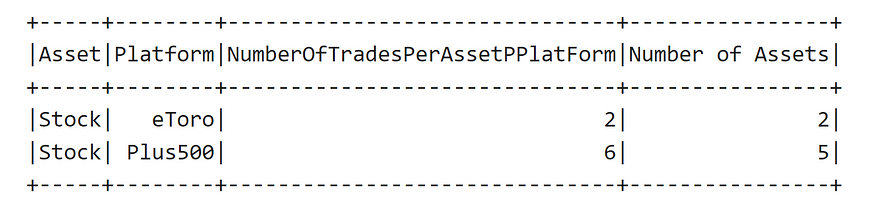


output — result of second approach

In the second approach, it can be seen that some of the columns are duplicated, so it is recommended to rename them before joining. We can repeat the second approach as follows to get rid of the repetitive columns:

// First renaming the columns in join and creating a new DF  
  
val numberOfAssets\_renamed = numberOfAssets  
.withColumnRenamed("Asset","AssetV2")  
.withColumnRenamed("Platform","PlatformV2")  
  
// performing the join with the new DF and dropping the renamed columns  
  
val joinedDF3 = numberOfTradesPerPlatform  
.join(numberOfAssets\_renamed,   
numberOfTradesPerPlatform.col("Asset") === numberOfAssets\_renamed.col("AssetV2")   
&& numberOfTradesPerPlatform.col("Platform") === numberOfAssets\_renamed.col("PlatformV2"))  
.drop("AssetV2","PlatformV2")  
  
joinedDF3.show()

In the above, it is possible to figure out that first the columns “Asset” and “Platform” are renamed to “AssetV2” and “PlatformV2”. Then the new DataFrame is used for the join, and finally, the renamed columns are dropped. Now, it should be possible to see the output without any duplicates:

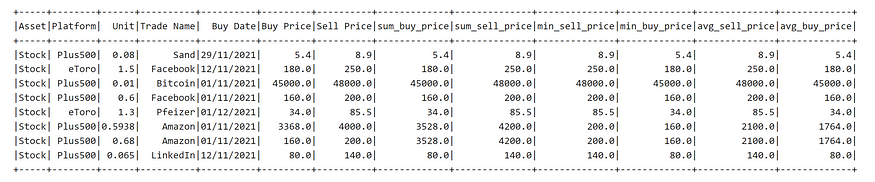


output — without duplicates

**Window Function — An alternative for groupBy()**

The window function allows one to perform certain calculations on specific dimensions of the data and add the result to the DataFrame. In fact, instead of performing a groupBy/aggregation and joining back the data with the original DataFrame, you still have the opportunity to perform the same by using the Window function. In case this is not clear, let us investigate it with an example:

import org.apache.spark.sql.expressions.Window  
  
val custom\_partition = Window.partitionBy($"Asset", $"Platform", $"Trade Name")  
  
val resulting\_df = DF.withColumn("sum\_buy\_price",   
sum("Buy Price").over(custom\_partition))  
.withColumn("sum\_sell\_price", sum("Sell Price").over(custom\_partition))  
.withColumn("min\_sell\_price", min("Sell Price").over(custom\_partition))  
.withColumn("min\_buy\_price", min("Buy Price").over(custom\_partition))  
.withColumn("avg\_sell\_price", avg("Sell Price").over(custom\_partition))  
.withColumn("avg\_buy\_price", avg("Buy Price").over(custom\_partition))  
  
  
resulting\_df.show()

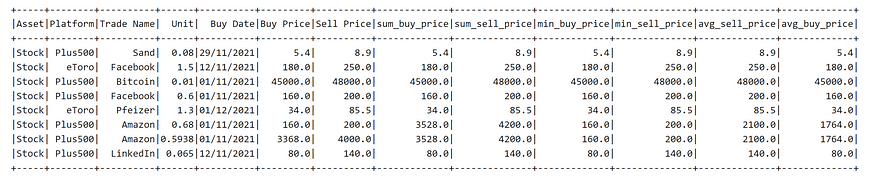


output of resulting\_df — Window function instead of groupBy/aggregation and join back

It is possible to see above that it seems like a groupBy/aggregation and joins back with the original DataFrame. Let us prove it by performing groupBy/aggregation and joining back and see if we can see the same result:

val groupBy\_aggregation\_df = DF.groupBy($"Asset", $"Platform", $"Trade Name")  
.agg(sum("Buy Price").as("sum\_buy\_price"),   
sum("Sell Price").as("sum\_sell\_price"),   
min("Buy Price").as("min\_buy\_price"),   
min("Sell Price").as("min\_sell\_price"),   
avg("Sell Price").as("avg\_sell\_price"),  
avg("Buy Price").as("avg\_buy\_price"))  
  
  
val joined\_back\_df = DF.join(groupBy\_aggregation\_df,   
Seq("Asset", "Platform","Trade Name"))  
  
joined\_back\_df.show()

As we can see in the following output, we get the exact same result as using the Window function:

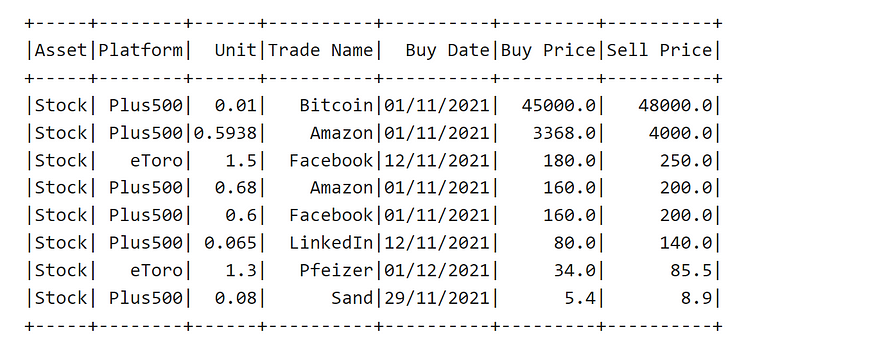


result of groupBy/aggregation and join back

**Spark orderBy()**

Normally, after a group by and aggregation, it may be needed to see the result in a specific order of a column although this operation is very time-consuming and the DataFrame resulting from the groupBy should not be too huge. However, it is also possible to apply orderBy() on the original DataFrame. As mentioned, if the DataFrame would be huge, this operation would not be recommended. Let us apply orderBy() on the DF:

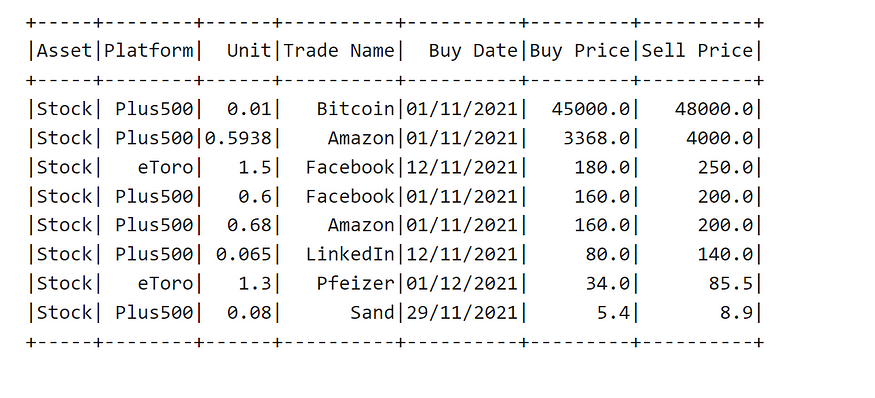
DF.orderBy(col("Buy Price").desc).show()



output — orderBy Buy Price descending

It is also possible to order by multiple columns and use *$* instead of *col()* like the following:

DF.orderBy(col("Buy Price").desc, $"Sell Price".desc).show()



output — order by multiple columns

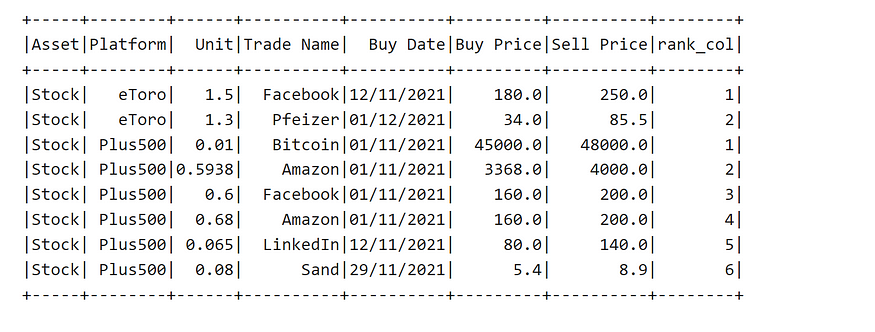
please use this link to join medium [*https://medium.com/@clever.tech.memes/membership*](https://medium.com/@clever.tech.memes/membership)*. Thank you for your great support.*

**Rank function**

Rank function together with Window allow to sort/rank the rows based on a specific column in our intended partition. Let us define a rank function by the help of *Window*and *orderBy*like the following:

import org.apache.spark.sql.expressions.Window  
  
val custom\_partition\_rank = Window.partitionBy($"Asset", $"Platform")  
 .orderBy($"Sell Price".desc)  
  
val resulting\_df = DF.withColumn("rank\_col",   
 row\_number().over(custom\_partition\_rank))  
  
  
resulting\_df.show()

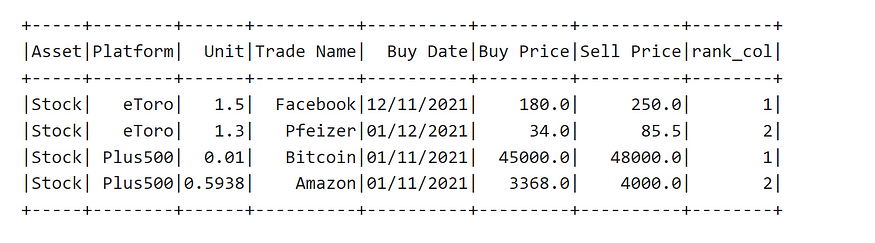
So, we can see that for each Platform, we can see the rank based on *Sell Price* in the following:



output — rank column

You can always keep the rows with your intended rank like the following in which we keep the rows with 1 or 2 rank using *.filter(col(“rank\_col”)<=2)*:

import org.apache.spark.sql.expressions.Window  
  
val custom\_partition\_rank = Window.partitionBy($"Asset", $"Platform")  
 .orderBy($"Sell Price".desc)  
  
val resulting\_df = DF.withColumn("rank\_col",   
 row\_number().over(custom\_partition\_rank))  
 .filter(col("rank\_col")<=2)  
  
  
resulting\_df.show()



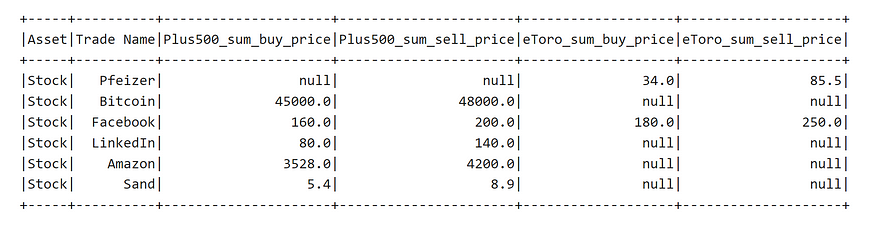
output — only keeping rows with 1 or 2 rank

**Pivoting in Spark**

Pivoting in Spark is a very much useful operation and you might require to pivot your data at some point. I have written a dedicated post regarding [pivoting in Spark](https://medium.com/@clever.tech.memes/spark-scala-pivoting-explained-a-crucial-topic-for-big-data-scientists-8e7d96733d16). Below, I briefly show on our example DF an example of pivoting:

val dfPrivot = DF.groupBy("Asset", "Trade Name").pivot("Platform",List("Plus500", "eToro"))  
.agg(sum("Buy Price").as("sum\_buy\_price"), sum("Sell Price").as("sum\_sell\_price"))  
  
dfPrivot.show()

In the above, pivot appeared after *groupBy()* and before *agg()* functions. Also, the input to the pivot is the column name i.e., *Platform*, and the corresponding list of values in the column name which is *Plus500* and *eToro*.



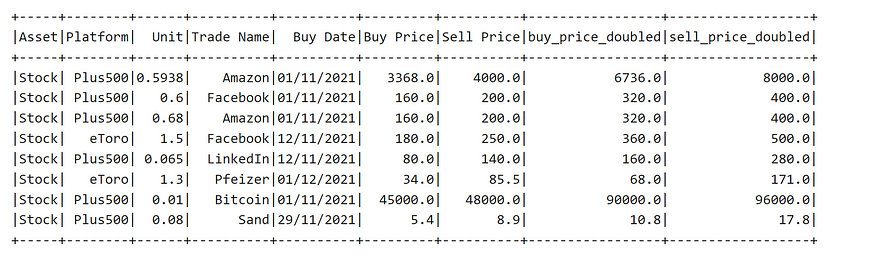
output — result of the pivoting example

**UDF in Spark**

UDFs are very much helpful to write a custom function that can be applied optimised on a Spark DataFrame. For defining a UDF, we need to a. import the “*org.apache.spark.sql.functions.udf” b.*define our own function*c.*wrap our function inside the*udf( \_) d.*and apply the UDF for creating a new column. Let us see how to use UDF in Spark with the following simple example:

import org.apache.spark.sql.functions.\_  
  
// Defining a simple multiplyByTwo function  
  
def multiplyByTwo(x: Float) : Float = x\*2  
  
// Wrap the function inside the udf( function\_name \_)  
  
val multiplyByTwoUDF = udf(multiplyByTwo \_)  
  
// Apply the UDF on our intended colum names  
val dfDoubled = DF.withColumn("buy\_price\_doubled",   
 multiplyByTwoUDF(col("Buy price")) )  
 .withColumn("sell\_price\_doubled",   
 multiplyByTwoUDF(col("Sell Price")))  
dfDoubled.show()

Let us see the output:

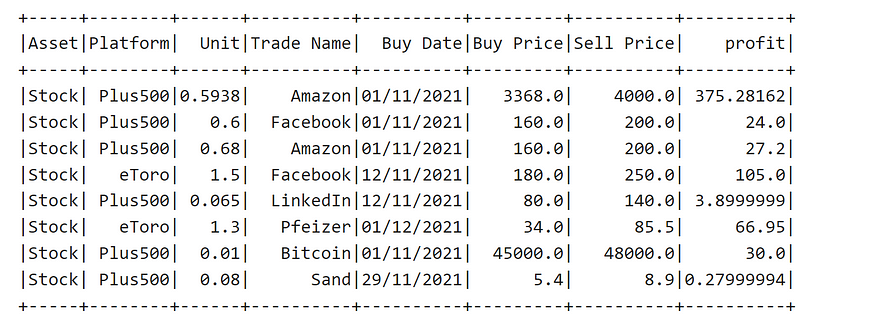


output — applying multiplyByTwoUDF for creating two new columns

Let us have a look at a more complicated example of using multiple inputs, slightly more complicated function and using of *$* sign for usage of UDFs:

def custom\_function(unit: Float, buy: Float, sell: Float) : Float = {  
   
 val diff = (sell - buy )   
   
 val res = unit \* diff  
   
 res  
}  
  
val custom\_udf = udf(custom\_function \_)  
  
val dfUDFExample = DF.withColumn("profit",   
custom\_udf($"Unit", $"Buy Price", $"Sell Price"))  
  
dfUDFExample.show()

This function has three inputs, and the corresponding UDF also accepts three parameters. So, the output looks like the following:



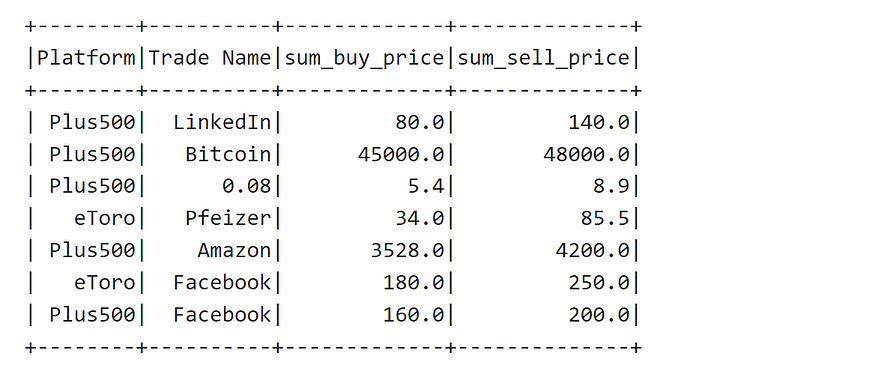
output — another example of UDF

**Spark SQL**

It is possible to write SQL queries and take advantage of the Spark distributed computing capability. For doing so, it is required to create a temporary view and apply the SQL query on the corresponding view. Let us show the usage of Spark SQL using our DataFrame DF:

val resulting\_df = spark.sql("SELECT Platform,   
`Trade Name`, sum(`Buy Price`) AS sum\_buy\_price,   
sum(`Sell Price`) AS sum\_sell\_price FROM DF\_View   
group by Platform, `Trade Name`")  
  
resulting\_df.show()

In the above, we use *spark.sql(…)*, and inside we write the SQL query. we group by Platformand *Trade Name* and calculate the sum of *Buy Price* and *Sell Price* in which the output is shown in the following.



output — Spark SQL

**Spark Input and Output**

Here, I would like to bring the code for reading and writing Parquet and CSV files in Spark

**Various approaches for reading Parquet files**

val df\_ = spark.read.parquet(file\_path)  
  
// Reading a list of paths with Parquet Files with same schemaVarious ways of reading CSV file  
  
val df = spark.read.parquet(path\_list:\_\*)

**Various approaches for reading CSV files**

// Reading csv files using a map of options  
  
val df\_csv\_v1 = spark.read.options(Map("inferSchema"->"true","delimiter"->","))  
.csv(file\_path)  
  
// Reading csv file in a single folder path  
  
val df\_csv\_v2 = spark.read.options(Map("inferSchema"->"true","delimiter"->","))  
.csv(folder\_path)  
  
// A list of file paths with same schema  
  
val df\_csv\_v3 = spark.read.options(Map("inferSchema"->"true","delimiter"->","))  
.csv(file\_path:\_\*)  
  
// An example with a single option  
  
val df\_csv\_v4 = spark.read.option("delimiter", ",").csv(file\_path)  
  
// Using option() sequentially  
  
val df\_csv\_v5 = spark.read.option("delimiter", ",").option("inferSchema, "true")  
.csv(file\_path)  
  
// Reading CSV file with a specific schema  
  
val df\_csv\_v6 = spark.read.format("csv").option("header", "true")  
 .schema(schema)  
 .load(file\_path)

**Writing Parquet and CSV Files**

// writes by the predefined number of partitions  
  
DF.spark.write.parquet(folder\_path)  
  
// defining some options and number of partitions using coalesce()  
  
DF.coalesce().spark.write.mode('append').parquet(folder\_path)  
  
  
// Using partition by  
  
DF.write.mode('append').partitionBy('col\_name').parquet(folder\_path)  
  
  
// writing CSV files  
  
DF.write.csv(folder\_path)

**Summary**

Spark is very much fundamental in today’s world of Big data. If you are a data scientist or data engineer, this cheat sheet can be helpful:

1. One requires to quickly refresh his/her Spark knowledge
2. Before an interview to review the Spark code snippet
3. For your daily job reference in case you are searching for specific syntax
4. Keeping the syntax active in mind by reviewing the cheat sheet