

# DataFrames in Spark (Part 1)

This course has been designed with Spark 2.0.1 in mind (Oct 2016), and was updated with some niceties and new style guide for Spark 2.1.0 in June 2017.

## 1. Overview

### 1.1. RDDs versus DataFrames

What is Spark SQL?

- Spark SQL takes basic RDDs and puts a schema on them.

What are schemas?

- Schema = Table Names + Column Names + Column Types

What are the pros of schemas?

- Schemas enable using column names instead of column positions
- Schemas enable queries using SQL and DataFrame syntax
- Schemas make your data more structured.

What is a DataFrame?

- DataFrames are the primary abstraction in Spark SQL.
- Think of a DataFrames as RDDs with schema.

What is a schema?

- Schemas are metadata about your data.
- Schemas define table names, column names, and column types over your data.
- Schemas enable using SQL and DataFrame syntax to query your RDDs, instead of using column positions.

## 2. Operational DataFrames in Python

We'll proceed along the usual spark flow (see above).

1. create the enviromnent to run Spark SQL from python

2. create DataFrames from RDDs or from files
3. run some transformations
4. execute actions to obtain values (local objects in python)

## 2.1. Initializing a SparkContext and SqlContext in Python

Using:

```
import pyspark as ps
sc = ps.SparkContext('local[4]')
```

will create a *"local"* cluster made of the driver using all 4 cores.

```
In [91]: # Get pyspark, spark
import findspark
findspark.init('/home/sparkles/spark-2.1.0-bin-hadoop2.7')
import pyspark

import pyspark as ps    # for the pyspark suite
import warnings         # for displaying warning
```

```
In [92]: try:
    # we try to create a SparkContext to work locally on all cpus available
    sc = ps.SparkContext('local[4]')
    print("Just created a SparkContext")
except ValueError:
    # give a warning if SparkContext already exists (for use inside pyspark)
    warnings.warn("SparkContext already exists in this scope")

/home/sparkles/.local/lib/python3.5/site-packages/ipykernel_launcher.py:7: UserWarning: SparkContext already exists in this scope
import sys
```

Then we create a SQLContext using our SparkContext as argument.

```
In [93]: sqlContext = ps.SQLContext(sc)
```

## 2.2. Creating a DataFrame

### 2.2.1. From an RDD (specifying schema)

You can create a DataFrame from an existing RDD (whatever source you used to create this one), if you add a schema.

To build a schema, you will use existing data types provided in the `pyspark.sql.types` (<https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#module-pyspark.sql.types>) module. Here's a list of the most useful ones (subjective criteria).

Types	Python-like type
StringType	string
IntegerType	int
FloatType	float
ArrayType*	array or list
MapType	dict

\* see later UDF functions on how to use that

```
In [94]: # remember that csv file ?
def casting_function(row):
    _id, date, store, state, product, amount = row
    return (int(_id), date, int(store), state, int(product), float(amount))

rdd_sales = sc.textFile('data/sales.csv')\
    .map(lambda rowstr : rowstr.split(","))\
    .filter(lambda row: not row[0].startswith('#'))\
    .map(casting_function)

rdd_sales.collect()
```

```
Out[94]: [(101, '11/13/2014', 100, 'WA', 331, 300.0),
(104, '11/18/2014', 700, 'OR', 329, 450.0),
(102, '11/15/2014', 203, 'CA', 321, 200.0),
(106, '11/19/2014', 202, 'CA', 331, 330.0),
(103, '11/17/2014', 101, 'WA', 373, 750.0),
(105, '11/19/2014', 202, 'CA', 321, 200.0)]
```

```
In [95]: rdd_sales
```

```
Out[95]: PythonRDD[332] at collect at <ipython-input-94-2cc849822204>:9
```

```
In [96]: rdd_sales1 = rdd_sales.map(lambda x : x)
```

```
In [97]: rdd_sales1
```

```
Out[97]: PythonRDD[333] at RDD at PythonRDD.scala:48
```

```
In [98]: # import the many data types
from pyspark.sql.types import (StructType,
    StructField, IntegerType, StringType, FloatType)

# create a schema of your own
schema = StructType( [
    StructField('id',IntegerType(),True),
    StructField('date',StringType(),True),
    StructField('store',IntegerType(),True),
    StructField('state',StringType(),True),
    StructField('product',IntegerType(),True),
    StructField('amount',FloatType(),True) ] )

# feed that into a DataFrame
df = sqlContext.createDataFrame(rdd_sales,schema)

# show the result
df.show()

# print the schema
df.printSchema()
```

```
+---+-----+-----+-----+-----+
| id|      date|store|state|product|amount|
+---+-----+-----+-----+-----+
|101|11/13/2014|  100|  WA|    331| 300.0|
|104|11/18/2014|  700|  OR|    329| 450.0|
|102|11/15/2014|  203|  CA|    321| 200.0|
|106|11/19/2014|  202|  CA|    331| 330.0|
|103|11/17/2014|  101|  WA|    373| 750.0|
|105|11/19/2014|  202|  CA|    321| 200.0|
+---+-----+-----+-----+-----+
```

```
root
|-- id: integer (nullable = true)
|-- date: string (nullable = true)
|-- store: integer (nullable = true)
|-- state: string (nullable = true)
|-- product: integer (nullable = true)
|-- amount: float (nullable = true)
```

## 2.2.2. Reading from files (inferring schema)

Use `sqlContext.read.csv`

(<https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrameReader.csv>)  
to load a CSV into a DataFrame. You can specify every useful parameter in there. It can infer the schema.

```
In [99]: # read CSV
df = sqlContext.read.csv('data/sales.csv',
                          header=True,      # use headers or not
                          quote='\"',      # char for quotes
                          sep=",",          # char for separation
                          inferSchema=True) # do we infer schema or not ?

# prints the schema
df.printSchema()

# some functions are still valid
print("line count: {}".format(df.count()))

# show the table in a oh-so-nice format
df.show()
```

```
root
|-- #ID: integer (nullable = true)
|-- Date: string (nullable = true)
|-- Store: integer (nullable = true)
|-- State: string (nullable = true)
|-- Product: integer (nullable = true)
|-- Amount: double (nullable = true)

line count: 6
+---+-----+-----+-----+-----+-----+
|#ID|      Date|Store|State|Product|Amount|
+---+-----+-----+-----+-----+-----+
|101|11/13/2014|  100|  WA|    331| 300.0|
|104|11/18/2014|  700|  OR|    329| 450.0|
|102|11/15/2014|  203|  CA|    321| 200.0|
|106|11/19/2014|  202|  CA|    331| 330.0|
|103|11/17/2014|  101|  WA|    373| 750.0|
|105|11/19/2014|  202|  CA|    321| 200.0|
+---+-----+-----+-----+-----+-----+
```

Use `sqlContext.read.json`

(<https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrameReader>)  
to load a JSON file into a DataFrame. You can specify every useful parameter in there. It can infer the schema.

```
In [100]: # read JSON
df = sqlContext.read.json('data/sales.json')

# prints the schema
df.printSchema()

# some functions are still valid
print("line count: {}".format(df.count()))

# show the table in a oh-so-nice format
df.show()
```

```
root
|-- amount: double (nullable = true)
|-- date: string (nullable = true)
|-- id: long (nullable = true)
|-- product: long (nullable = true)
|-- state: string (nullable = true)
|-- store: long (nullable = true)

line count: 6
+-----+-----+---+-----+-----+-----+
|amount|      date| id|product|state|store|
+-----+-----+---+-----+-----+-----+
| 300.0|11/13/2014|101|   331|  WA|  100|
| 450.0|11/18/2014|104|   329|  OR|  700|
| 200.0|11/15/2014|102|   321|  CA|  203|
| 330.0|11/19/2014|106|   331|  CA|  202|
| 750.0|11/17/2014|103|   373|  WA|  101|
| 200.0|11/19/2014|105|   321|  CA|  202|
+-----+-----+---+-----+-----+-----+
```

```
In [101]: # read JSON
df = sqlContext.read.json('data/sales2.json.gz')

# show the table in a oh-so-nice format
df.show()
```

amount	date	id	product	state	store
300.0	11/13/2014	101	331	WA	100
450.0	11/18/2014	104	329	OR	700
200.0	11/15/2014	102	321	CA	203
330.0	11/19/2014	106	331	CA	202
750.0	11/17/2014	103	373	WA	101
200.0	11/19/2014	105	321	CA	202

## Niceties

```
In [102]: # read JSON
df = sqlContext.read.json('data/sales.json')
```

```
In [103]: df.show()
```

amount	date	id	product	state	store
300.0	11/13/2014	101	331	WA	100
450.0	11/18/2014	104	329	OR	700
200.0	11/15/2014	102	321	CA	203
330.0	11/19/2014	106	331	CA	202
750.0	11/17/2014	103	373	WA	101
200.0	11/19/2014	105	321	CA	202

```
In [ ]:
```

```
In [104]: df.printSchema()
```

```
root
|-- amount: double (nullable = true)
|-- date: string (nullable = true)
|-- id: long (nullable = true)
|-- product: long (nullable = true)
|-- state: string (nullable = true)
|-- store: long (nullable = true)
```

```
In [105]: df.columns
```

```
Out[105]: ['amount', 'date', 'id', 'product', 'state', 'store']
```

```
In [106]: df.describe()
```

```
Out[106]: DataFrame[summary: string, amount: string, date: string, id: string, product: string, state: string, store: string]
```

```
In [107]: df.describe().show()
```

```
+-----+-----+-----+-----+-----+
|summary|          amount|      date|          id|          p
|product|state|          store|
+-----+-----+-----+-----+-----+
| count|          6|          6|          6|
6|    6|          6|
| mean| 371.6666666666667| null| 103.5| 334.3333333
333333| null|251.33333333333334|
| stddev|207.40459654179958| null|1.8708286933869716|19.500427345
744672| null|225.39180700874346|
| min|          200.0|11/13/2014|          101|
321|    CA|          100|
| max|          750.0|11/19/2014|          106|
373|    WA|          700|
+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+
```

Inferred Schema (refresher)



```
In [108]: # read CSV
df_csv = sqlContext.read.csv('data/sales.csv',
                              header=True,      # use headers or not
                              quote='"',        # char for quotes
                              sep=",",         # char for separation
                              inferSchema=True) # do we infer schema or not ?

# prints the schema
df_csv.printSchema()
df_csv.show()
```

```
root
|-- #ID: integer (nullable = true)
|-- Date: string (nullable = true)
|-- Store: integer (nullable = true)
|-- State: string (nullable = true)
|-- Product: integer (nullable = true)
|-- Amount: double (nullable = true)
```

#ID	Date	Store	State	Product	Amount
101	11/13/2014	100	WA	331	300.0
104	11/18/2014	700	OR	329	450.0
102	11/15/2014	203	CA	321	200.0
106	11/19/2014	202	CA	331	330.0
103	11/17/2014	101	WA	373	750.0
105	11/19/2014	202	CA	321	200.0

Manual Schema, with latest recommended style as of 2.1.0

```
In [109]: # Latest recommended flow as of 2.1.0
import findspark
findspark.init('/home/sparkles/spark-2.1.0-bin-hadoop2.7')
import pyspark

from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('Lecture').getOrCreate() # called '

from pyspark.sql.types import (StructType,
                                StructField, IntegerType, StringType, FloatType)

data_schema = [StructField('#ID', IntegerType(), True),
                StructField('Date', StringType(), True),
                StructField('Store', IntegerType(), True),
                StructField('State', StringType(), True),
                StructField('Product', IntegerType(), True),
```

```

        StructField('Amount', FloatType(), True)]

schema = StructType(fields=data_schema)

df = spark.read.csv('data/sales.csv',
                    header=True,
                    quote='\"',
                    sep=",",
                    schema=schema)

# compare to:
# read CSV
# df_csv = sqlContext.read.csv('data/sales.csv',
#                               header=True,      # use headers or not
#                               quote='\"',      # char for quotes
#                               sep=",",        # char for separation
#                               inferSchema=True) # do we infer schema or not

# prints the schema

df.printSchema()
df.show()

```

```

root
|-- #ID: integer (nullable = true)
|-- Date: string (nullable = true)
|-- Store: integer (nullable = true)
|-- State: string (nullable = true)
|-- Product: integer (nullable = true)
|-- Amount: float (nullable = true)

```

#ID	Date	Store	State	Product	Amount
101	11/13/2014	100	WA	331	300.0
104	11/18/2014	700	OR	329	450.0
102	11/15/2014	203	CA	321	200.0
106	11/19/2014	202	CA	331	330.0
103	11/17/2014	101	WA	373	750.0
105	11/19/2014	202	CA	321	200.0

```
In [110]: df.show()
```

```
+---+-----+-----+-----+-----+
|#ID|      Date|Store|State|Product|Amount|
+---+-----+-----+-----+-----+
|101|11/13/2014|  100|  WA|    331| 300.0|
|104|11/18/2014|  700|  OR|    329| 450.0|
|102|11/15/2014|  203|  CA|    321| 200.0|
|106|11/19/2014|  202|  CA|    331| 330.0|
|103|11/17/2014|  101|  WA|    373| 750.0|
|105|11/19/2014|  202|  CA|    321| 200.0|
+---+-----+-----+-----+-----+
```

```
In [111]: df.collect()
```

```
Out[111]: [Row(#ID=101, Date='11/13/2014', Store=100, State='WA', Product=331, Amount=300.0),
           Row(#ID=104, Date='11/18/2014', Store=700, State='OR', Product=329, Amount=450.0),
           Row(#ID=102, Date='11/15/2014', Store=203, State='CA', Product=321, Amount=200.0),
           Row(#ID=106, Date='11/19/2014', Store=202, State='CA', Product=331, Amount=330.0),
           Row(#ID=103, Date='11/17/2014', Store=101, State='WA', Product=373, Amount=750.0),
           Row(#ID=105, Date='11/19/2014', Store=202, State='CA', Product=321, Amount=200.0)]
```

```
In [112]: type(df['State'])
```

```
Out[112]: pyspark.sql.column.Column
```

```
In [113]: df.select('State')
```

```
Out[113]: DataFrame[State: string]
```

```
In [114]: df.select('State').show()
```

```
+-----+
|State|
+-----+
|    WA|
|    OR|
|    CA|
|    CA|
|    WA|
|    CA|
+-----+
```

```
In [115]: df.head(2)
```

```
Out[115]: [Row(#ID=101, Date='11/13/2014', Store=100, State='WA', Product=331, Amount=300.0),
           Row(#ID=104, Date='11/18/2014', Store=700, State='OR', Product=329, Amount=450.0)]
```

```
In [116]: df.head(2)[0]
```

```
Out[116]: Row(#ID=101, Date='11/13/2014', Store=100, State='WA', Product=331, Amount=300.0)
```

```
In [117]: df.head(1)[0]
```

```
Out[117]: Row(#ID=101, Date='11/13/2014', Store=100, State='WA', Product=331, Amount=300.0)
```

```
In [118]: df.select(['State', 'Amount']).show()
```

```
+-----+-----+
|State|Amount|
+-----+-----+
|    WA| 300.0|
|    OR| 450.0|
|    CA| 200.0|
|    CA| 330.0|
|    WA| 750.0|
|    CA| 200.0|
+-----+-----+
```

```
In [119]: df.withColumn('newState', df['State']).show()
```

#ID	Date	Store	State	Product	Amount	newState
101	11/13/2014	100	WA	331	300.0	WA
104	11/18/2014	700	OR	329	450.0	OR
102	11/15/2014	203	CA	321	200.0	CA
106	11/19/2014	202	CA	331	330.0	CA
103	11/17/2014	101	WA	373	750.0	WA
105	11/19/2014	202	CA	321	200.0	CA

```
In [120]: df.withColumnRenamed('State', 'newState').show()
```

#ID	Date	Store	newState	Product	Amount
101	11/13/2014	100	WA	331	300.0
104	11/18/2014	700	OR	329	450.0
102	11/15/2014	203	CA	321	200.0
106	11/19/2014	202	CA	331	330.0
103	11/17/2014	101	WA	373	750.0
105	11/19/2014	202	CA	321	200.0

I hear you like SQL...

```
In [121]: df.createOrReplaceTempView('sales') # 'sales' is the name of the 'table'
```

```
In [122]: results = spark.sql("SELECT * FROM sales")
```

```
In [123]: results.show()
```

#ID	Date	Store	State	Product	Amount
101	11/13/2014	100	WA	331	300.0
104	11/18/2014	700	OR	329	450.0
102	11/15/2014	203	CA	321	200.0
106	11/19/2014	202	CA	331	330.0
103	11/17/2014	101	WA	373	750.0
105	11/19/2014	202	CA	321	200.0

```
In [124]: results = spark.sql("SELECT * FROM sales WHERE Product=331")
          results.show()
```

```
+---+-----+-----+-----+-----+-----+
|#ID|      Date|Store|State|Product|Amount|
+---+-----+-----+-----+-----+-----+
|101|11/13/2014|  100|   WA|    331| 300.0|
|106|11/19/2014|  202|   CA|    331| 330.0|
+---+-----+-----+-----+-----+-----+
```

```
In [ ]:
```

```
In [ ]:
```

How could I do that using DataFrame syntax?

```
In [125]: results = df.filter(df['Product'] == 331)
          results.show()
```

```
+---+-----+-----+-----+-----+-----+
|#ID|      Date|Store|State|Product|Amount|
+---+-----+-----+-----+-----+-----+
|101|11/13/2014|  100|   WA|    331| 300.0|
|106|11/19/2014|  202|   CA|    331| 330.0|
+---+-----+-----+-----+-----+-----+
```

```
In [126]: results = df.filter(df['Product'] == 331).collect()
          row = results[0]
```

```
In [127]: row.asDict()
```

```
Out[127]: {'#ID': 101,
           'Amount': 300.0,
           'Date': '11/13/2014',
           'Product': 331,
           'State': 'WA',
           'Store': 100}
```

```
In [128]: row.asDict()['State']
```

```
Out[128]: 'WA'
```

```
In [129]: res = df.filter( (df['Product'] == 331) & ~(df['State'] == 'WA') ).collect()
[r.asDict() for r in res]
```

```
Out[129]: [{'#ID': 106,
            'Amount': 330.0,
            'Date': '11/19/2014',
            'Product': 331,
            'State': 'CA',
            'Store': 202}]
```

## 2.3. Actions : turning your DataFrame into a local object

Some actions just remain the same, you won't have to learn Spark all over again.

Some new actions give you the possibility to describe and show the content in a more fashionable manner.

When used/executed in IPython or in a notebook, they **launch the processing of the DAG**. This is where Spark stops being **lazy**. This is where your script will take time to execute.

	Method	DF, RDD
<a href="https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrame.collect">.collect()</a>	<a href="https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrame.collect">.collect()</a>	identical
<a href="https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrame.count">.count()</a>	<a href="https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrame.count">.count()</a>	identical
<a href="https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrame.take">.take(n)</a>	<a href="https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrame.take">.take(n)</a>	identical
<a href="https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrame.top">.top(n)</a>	<a href="https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrame.top">.top(n)</a>	identical
<a href="https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrame.first">.first()</a>	<a href="https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrame.first">.first()</a>	identical
<a href="https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrame.show">.show(n)</a>	<a href="https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrame.show">.show(n)</a>	new
<a href="https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrame.toPandas">.toPandas()</a>	<a href="https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrame.toPandas">.toPandas()</a>	new

`.printSchema(*cols)`  
<https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrame.printSchema>\* ne

`.describe(*cols)`  
<https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrame.describe> ne

`.sum(*cols)`  
<https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.GroupedData.sum> differe

`.mean(*cols)`  
<https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.GroupedData.mean> differe

`.min(*cols)`  
<https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.GroupedData.min> differe

`.max(*cols)`  
<https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.GroupedData.max> differe

```
In [130]: # read CSV
df_sales = sqlContext.read.csv('data/sales.csv',
                                header=True,          # use headers or not
                                quote='\"',           # char for quotes
                                sep=",",              # char for separation
                                inferSchema=True)     # do we infer schema or not ?
```

```
In [131]: df_sales.show()
```

#ID	Date	Store	State	Product	Amount
101	11/13/2014	100	WA	331	300.0
104	11/18/2014	700	OR	329	450.0
102	11/15/2014	203	CA	321	200.0
106	11/19/2014	202	CA	331	330.0
103	11/17/2014	101	WA	373	750.0
105	11/19/2014	202	CA	321	200.0



```
In [132]: df_sales.toPandas()
```

```
Out[132]:
```

	#ID	Date	Store	State	Product	Amount
0	101	11/13/2014	100	WA	331	300.0
1	104	11/18/2014	700	OR	329	450.0
2	102	11/15/2014	203	CA	321	200.0
3	106	11/19/2014	202	CA	331	330.0
4	103	11/17/2014	101	WA	373	750.0
5	105	11/19/2014	202	CA	321	200.0

This is how `.collect()` returns things...

```
In [133]: df_sales.collect()
```

```
Out[133]: [Row(#ID=101, Date='11/13/2014', Store=100, State='WA', Product=331, Amount=300.0),
  Row(#ID=104, Date='11/18/2014', Store=700, State='OR', Product=329, Amount=450.0),
  Row(#ID=102, Date='11/15/2014', Store=203, State='CA', Product=321, Amount=200.0),
  Row(#ID=106, Date='11/19/2014', Store=202, State='CA', Product=331, Amount=330.0),
  Row(#ID=103, Date='11/17/2014', Store=101, State='WA', Product=373, Amount=750.0),
  Row(#ID=105, Date='11/19/2014', Store=202, State='CA', Product=321, Amount=200.0)]
```

```
In [134]: # prints the schema
print("--- printSchema()")
df_sales.printSchema()

# prints the table itself
print("--- show()")
df_sales.show()

# show the statistics of all numerical columns
print("--- describe()")
df_sales.describe().show()

# show the statistics of one specific column
print("--- describe(Amount)")
df_sales.describe("Amount").show()

--- printSchema()
```

```

root
|-- #ID: integer (nullable = true)
|-- Date: string (nullable = true)
|-- Store: integer (nullable = true)
|-- State: string (nullable = true)
|-- Product: integer (nullable = true)
|-- Amount: double (nullable = true)

--- show()
+---+-----+-----+-----+-----+-----+
|#ID|      Date|Store|State|Product|Amount|
+---+-----+-----+-----+-----+-----+
|101|11/13/2014|  100|  WA|   331| 300.0|
|104|11/18/2014|  700|  OR|   329| 450.0|
|102|11/15/2014|  203|  CA|   321| 200.0|
|106|11/19/2014|  202|  CA|   331| 330.0|
|103|11/17/2014|  101|  WA|   373| 750.0|
|105|11/19/2014|  202|  CA|   321| 200.0|
+---+-----+-----+-----+-----+-----+

--- describe()
+-----+-----+-----+-----+-----+-----+
|summary|          #ID|      Date|      Store|State|
Product|          Amount|
+-----+-----+-----+-----+-----+-----+
| count|          6|          6|          6|  6|
6|          6|
| mean|        103.5|      null|251.33333333333334| null| 334.3
333333333333| 371.66666666666667|
| stddev|1.8708286933869716|      null|225.39180700874346| null|19.500
427345744672|207.40459654179958|
| min|          101|11/13/2014|          100|  CA|
321|          200.0|
| max|          106|11/19/2014|          700|  WA|
373|          750.0|
+-----+-----+-----+-----+-----+-----+

--- describe(Amount)
+-----+-----+
|summary|      Amount|
+-----+-----+
| count|          6|
| mean| 371.66666666666667|
| stddev|207.40459654179958|
| min|          200.0|
| max|          750.0|
+-----+-----+

```

## 2.3. Transformations on DataFrames

- They are still **lazy**: Spark doesn't apply the transformation right away, it just builds on the **DAG**
- They transform a DataFrame into another because DataFrames are also **immutable**.
- They can be **wide** or **narrow** (whether they shuffle partitions or not).

You got that... DataFrames are just RDDs with a schema.

	Method	Type
<a href="http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD.map">.map(func)</a>		transformation
<a href="http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD.flatMap">.flatMap(func)</a>		transformation
<a href="http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD.filter">.filter(func)</a>		transformation
<a href="http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD.sample">.sample()</a>		transformation
<a href="http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD.distinct">.distinct()</a>		transformation
<a href="http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD.keys">.keys()</a>		transformation

<http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD.values> `.values()` transformation

<http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD.join> `.join(rddB)` transformation

<http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD.reduceByKey> `.reduceByKey()` transformation

<http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD.groupByKey> `.groupByKey()` transformation

<http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD.sortBy> `.sortBy(keyfunc)` transformation

<http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD.sortByKey> `.sortByKey()` transformation sorti

### 2.3.2. `.withColumn()`: adding column using operations or functions

**.withColumn("label", func):**

```
In [135]: # read CSV
df_aapl = sqlContext.read.csv('data/aapl.csv',
                                header=True,           # use headers or not
                                quote='\"',           # char for quotes
                                sep=",",             # char for separation
                                inferSchema=True)      # do we infer schema or not ?

df_aapl.show(5)

df_aapl.printSchema()
```

```
+-----+-----+-----+-----+-----+-----+
+-----+
|          Date|      Open|      High|      Low|      Close|  Vo
lume| Adj Close|
+-----+-----+-----+-----+-----+-----+
+-----+
|2016-10-25 00:00:...|117.949997|118.360001|117.309998|      118.25|3919
0300|      118.25|
|2016-10-24 00:00:...|117.099998|117.739998|      117.0|117.650002|2353
8700|117.650002|
|2016-10-21 00:00:...|116.809998|116.910004|116.279999|116.599998|2319
2700|116.599998|
|2016-10-20 00:00:...|116.860001|117.379997|116.330002|117.059998|2412
5800|117.059998|
|2016-10-19 00:00:...|      117.25|117.760002|113.800003|117.120003|2003
4600|117.120003|
+-----+-----+-----+-----+-----+-----+
+-----+
```

only showing top 5 rows

```
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)
```

**.withColumn("label", func): constant value**

```
In [136]: from pyspark.sql.functions import lit
```

```
df_out = df_aapl.withColumn("blabla", lit("echo"))
```

```
df_out.show(5)
```

```
+-----+-----+-----+-----+-----+-----+
----+-----+-----+
|          Date|      Open|      High|      Low|      Close|  Vo
lume| Adj Close|blabla|
+-----+-----+-----+-----+-----+-----+
----+-----+-----+
|2016-10-25 00:00:...|117.949997|118.360001|117.309998|      118.25|3919
0300|      118.25|  echo|
|2016-10-24 00:00:...|117.099998|117.739998|      117.0|117.650002|2353
8700|117.650002|  echo|
|2016-10-21 00:00:...|116.809998|116.910004|116.279999|116.599998|2319
2700|116.599998|  echo|
|2016-10-20 00:00:...|116.860001|117.379997|116.330002|117.059998|2412
5800|117.059998|  echo|
|2016-10-19 00:00:...|      117.25|117.760002|113.800003|117.120003|2003
4600|117.120003|  echo|
+-----+-----+-----+-----+-----+-----+
----+-----+-----+
only showing top 5 rows
```

**.withColumn("label", func) : column operations**

```
In [137]: df_out = df_aapl.withColumn("diff", df_aapl.High - df_aapl.Low)

df_out.show(5)
```

```
+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+
|           Date|      Open|      High|      Low|      Close|  Vo
lume| Adj Close|           diff|
+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+
|2016-10-25 00:00:...|117.949997|118.360001|117.309998|      118.25|3919
0300|      118.25|1.05000300000000038|
|2016-10-24 00:00:...|117.099998|117.739998|      117.0|117.650002|2353
8700|117.650002|0.73999799999999999|
|2016-10-21 00:00:...|116.809998|116.910004|116.279999|116.599998|2319
2700|116.599998| 0.63000499999999997|
|2016-10-20 00:00:...|116.860001|117.379997|116.330002|117.059998|2412
5800|117.059998|1.04999500000000098|
|2016-10-19 00:00:...|      117.25|117.760002|113.800003|117.120003|2003
4600|117.120003|3.95999899999999963|
+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+
only showing top 5 rows
```

**.withColumn("label", func) : user defined function**

```
In [138]: from pyspark.sql.functions import udf
from pyspark.sql.types import DoubleType

def my_specialfunc(h,l,o,c):
    return ((h-l)*(o-c))

my_specialfunc_udf = udf(lambda h,l,o,c : my_specialfunc(h,l,o,c), DoubleType)

df_out = df_aapl.withColumn("special", my_specialfunc_udf(df_aapl.High, df_aapl.Low, df_aapl.Open, df_aapl.Close))

df_out.show()
```

```
+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+
|           Date|      Open|      High|      Low|      Close|  Vo
lume| Adj Close|           special|
+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+
|2016-10-25 00:00:...|117.949997|118.360001|117.309998|      118.25|3919
0300|      118.25| -0.3150040500090051|
|2016-10-24 00:00:...|117.099998|117.739998|      117.0|117.650002|2353
```

```

8700|117.650002| -0.4070018599920009|
|2016-10-21 00:00:...|116.809998|116.910004|116.279999|116.599998|2319
2700|116.599998| 0.13230104999999545|
|2016-10-20 00:00:...|116.860001|117.379997|116.330002|117.059998|2412
5800|117.059998|-0.20999585001499796|
|2016-10-19 00:00:...| 117.25|117.760002|113.800003|117.120003|2003
4600|117.120003| 0.5147879900030115|
|2016-10-18 00:00:...| 118.18|118.209999|117.449997|117.470001|2455
3500|117.470001| 0.539600659998008|
|2016-10-17 00:00:...|117.330002|117.839996|116.779999|117.550003|2362
4900|117.550003|-0.23320039999701023|
|2016-10-14 00:00:...|117.879997|118.169998|117.129997|117.629997|3565
2200|117.629997| 0.26000025000000093|
|2016-10-13 00:00:...|116.790001|117.440002|115.720001|116.980003|3519
2400|116.980003| -0.3268036300019894|
|2016-10-12 00:00:...|117.349998|117.980003| 116.75|117.339996|3758
6800|117.339996|0.012302490006000045|
|2016-10-11 00:00:...|117.699997|118.690002|116.199997|116.300003|6404
1000|116.300003| 3.485992059969996|
|2016-10-10 00:00:...|115.019997| 116.75|114.720001|116.050003|3623
6000|116.050003| -2.090911149994004|
|2016-10-07 00:00:...|114.309998|114.559998|113.510002|114.059998|2435
8400|114.059998| 0.26249899999999826|
|2016-10-06 00:00:...|113.699997|114.339996|113.129997|113.889999|2877
9300|113.889999|-0.22990222999800763|
|2016-10-05 00:00:...|113.400002|113.660004|112.690002|113.050003|2145
3100|113.050003| 0.33949972999799477|
|2016-10-04 00:00:...|113.059998|114.309998|112.629997| 113.0|2973
6800| 113.0| 0.10079669999798783|
|2016-10-03 00:00:...|112.709999|113.050003|112.279999|112.519997|2170
1800|112.519997| 0.14630230000799438|
|2016-09-30 00:00:...|112.459999|113.370003|111.800003|113.050003|3637
9100|113.050003| -0.92630628000000078|
|2016-09-29 00:00:...|113.160004|113.800003|111.800003| 112.18|3588
7000| 112.18| 1.96000799999999878|
|2016-09-28 00:00:...|113.690002|114.639999| 113.43|113.949997|2964
1100|113.949997| -0.3145936900049861|
+-----+-----+-----+-----+-----+-----+
----+-----+-----+
only showing top 20 rows

```

**.select(\*cols) : selecting specific columns**



```
In [139]: df_out = df_aapl.select(["Open", "Close"])

df_out.show(5)
```

```
+-----+-----+
|      Open|      Close|
+-----+-----+
|117.949997|    118.25|
|117.099998|117.650002|
|116.809998|116.599998|
|116.860001|117.059998|
|    117.25|117.120003|
+-----+-----+
only showing top 5 rows
```

### **.groupBy(): aggregating in DataFrames**

```
In [140]: from pyspark.sql import functions as F

df_out = df_sales.groupBy("State").agg(F.sum("Amount"))

df_out.show()
```

```
+-----+-----+
|State|sum(Amount)|
+-----+-----+
|   OR|      450.0|
|   CA|      730.0|
|   WA|     1050.0|
+-----+-----+
```

### **.orderBy(): sorting by a column**

```
In [141]: df_out = df_sales.groupBy("State").agg(F.sum("Amount")).orderBy("sum(Amount)")
df_out.show()
```

```
+-----+-----+
| State | sum(Amount) |
+-----+-----+
|    WA |      1050.0 |
|    CA |       730.0 |
|    OR |       450.0 |
+-----+-----+
```

## 3. Let's design chains of transformations together ! (reloaded)

### 3.1. Computing sales per state

#### Input DataFrame

```
In [142]: # read CSV
df_sales = sqlContext.read.csv('data/sales.csv',
                                header=True,          # use headers or not
                                quote='"',            # char for quotes
                                sep=",",              # char for separation
                                inferSchema=True)     # do we infer schema or not ?

df_sales.show()
```

```
+---+-----+-----+-----+-----+-----+
|#ID|      Date|Store|State|Product|Amount|
+---+-----+-----+-----+-----+-----+
|101|11/13/2014|  100|  WA|    331|  300.0|
|104|11/18/2014|  700|  OR|    329|  450.0|
|102|11/15/2014|  203|  CA|    321|  200.0|
|106|11/19/2014|  202|  CA|    331|  330.0|
|103|11/17/2014|  101|  WA|    373|  750.0|
|105|11/19/2014|  202|  CA|    321|  200.0|
+---+-----+-----+-----+-----+-----+
```

#### Task

You want to obtain a sorted ~~RDD~~ DataFrame of the states in which you have most sales done (amount).

What transformations do you need to apply ? If you had to draw a workflow of the transformations to apply ?

## Code

```
In [143]: df_out = df_sales
          df_out.show()
```

#ID	Date	Store	State	Product	Amount
101	11/13/2014	100	WA	331	300.0
104	11/18/2014	700	OR	329	450.0
102	11/15/2014	203	CA	321	200.0
106	11/19/2014	202	CA	331	330.0
103	11/17/2014	101	WA	373	750.0
105	11/19/2014	202	CA	321	200.0

## Solution (use your mouse to uncover)

```
df_out = df_sales.groupBy(df_sales.State)\
    .agg(F.sum(df_sales.Amount).alias('Money'))\
    .orderBy("Money", ascending=False)

df_out.show()
```

```
In [144]: # revealed solution here...
```

## 3.2. Find the date on which AAPL's stock price was the highest

### Input DataFrame

```
In [145]: # read CSV
df_aapl = sqlContext.read.csv('data/aapl.csv',
                                header=True,          # use headers or not
                                quote='"',            # char for quotes
                                sep=",",              # char for separation
                                inferSchema=True)      # do we infer schema or not ?

df_aapl.show(5)
```

```
+-----+-----+-----+-----+-----+-----+
+-----+
|          Date|      Open|      High|      Low|      Close|  Vo
lume| Adj Close|
+-----+-----+-----+-----+-----+-----+
+-----+
|2016-10-25 00:00:...|117.949997|118.360001|117.309998|      118.25|3919
0300|      118.25|
|2016-10-24 00:00:...|117.099998|117.739998|      117.0|117.650002|2353
8700|117.650002|
|2016-10-21 00:00:...|116.809998|116.910004|116.279999|116.599998|2319
2700|116.599998|
|2016-10-20 00:00:...|116.860001|117.379997|116.330002|117.059998|2412
5800|117.059998|
|2016-10-19 00:00:...|      117.25|117.760002|113.800003|117.120003|2003
4600|117.120003|
+-----+-----+-----+-----+-----+-----+
+-----+
only showing top 5 rows
```

## Task

Now, design a pipeline that would :

1. ~~filter out headers and last line~~
2. ~~split each line based on comma~~
3. keep only fields for Date (~~col 0~~) and Close (~~col 4~~)
4. order by Close in descending order

## Code

```
In [146]: df_out = df_aapl # apply transformation here...

df_out.show(5)
```

```
+-----+-----+-----+-----+-----+-----+
+-----+
|           Date|         Open|         High|         Low|         Close|  Vo
lume| Adj Close|
+-----+-----+-----+-----+-----+-----+
+-----+
|2016-10-25 00:00:...|117.949997|118.360001|117.309998|         118.25|3919
0300|         118.25|
|2016-10-24 00:00:...|117.099998|117.739998|         117.0|117.650002|2353
8700|117.650002|
|2016-10-21 00:00:...|116.809998|116.910004|116.279999|116.599998|2319
2700|116.599998|
|2016-10-20 00:00:...|116.860001|117.379997|116.330002|117.059998|2412
5800|117.059998|
|2016-10-19 00:00:...|         117.25|117.760002|113.800003|117.120003|2003
4600|117.120003|
+-----+-----+-----+-----+-----+-----+
+-----+
only showing top 5 rows
```

## Solution

```
df_out.select("Close", "Date").orderBy(df_aapl.Close,
ascending=False).show(5)
```

```
In [147]: # revealed solution here...
```

## 4. Machine Learning on DataFrames

<http://spark.apache.org/docs/latest/ml-features.html> (<http://spark.apache.org/docs/latest/ml-features.html>)

```
In [148]: # read CSV
df_aapl = sqlContext.read.csv('data/aapl.csv',
                              header=True,      # use headers or not
                              quote='\"',        # char for quotes
                              sep=',',          # char for separation
                              inferSchema=True)  # do we infer schema or not ?

df_aapl.show(5)

df_aapl.printSchema()
```

```
+-----+-----+-----+-----+-----+-----+
+-----+
|          Date|      Open|      High|      Low|      Close|  Vo
lume| Adj Close|
+-----+-----+-----+-----+-----+-----+
+-----+
|2016-10-25 00:00:...|117.949997|118.360001|117.309998|      118.25|3919
0300|      118.25|
|2016-10-24 00:00:...|117.099998|117.739998|      117.0|117.650002|2353
8700|117.650002|
|2016-10-21 00:00:...|116.809998|116.910004|116.279999|116.599998|2319
2700|116.599998|
|2016-10-20 00:00:...|116.860001|117.379997|116.330002|117.059998|2412
5800|117.059998|
|2016-10-19 00:00:...|      117.25|117.760002|113.800003|117.120003|2003
4600|117.120003|
```

```
+-----+-----+-----+-----+-----+-----+
+-----+
only showing top 5 rows
```

```
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)
```

```
In [149]: from pyspark.ml.feature import MinMaxScaler, VectorAssembler

# assemble values in a vector
vectorAssembler = VectorAssembler(inputCols=["Close"],
                                   outputCol="features")

df_vector = vectorAssembler.transform(df_aapl)

scaler = MinMaxScaler(inputCol="features", outputCol="scaledfeatures")

# Compute summary statistics and generate MinMaxScalerModel
scalerModel = scaler.fit(df_vector)

# rescale each feature to range [min, max].
scaledData = scalerModel.transform(df_vector)
scaledData.select("features", "scaledfeatures").show(5)
```

features	scaledfeatures
[118.25]	[0.865963404782699]
[117.650002]	[0.8473472730564975]
[116.599998]	[0.8147688098332226]
[117.059998]	[0.8290412250646944]
[117.120003]	[0.8309029995776607]

only showing top 5 rows

In [ ]: