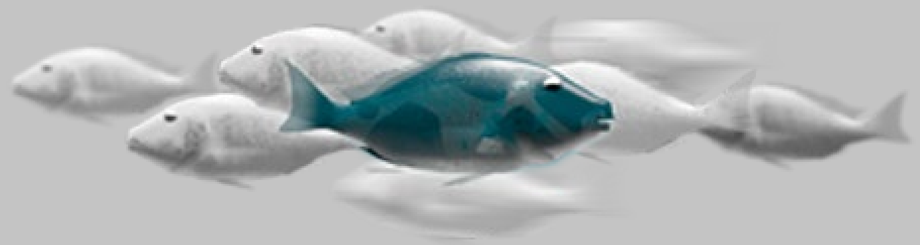
EDEM



Master Data Analytics

Apache Spark Sessions 2023-24
Spark SQL
Pablo Pons Roger

1. Apache Spark Data Structures: RDDs

Resilient Distributed Datasets (RDD)

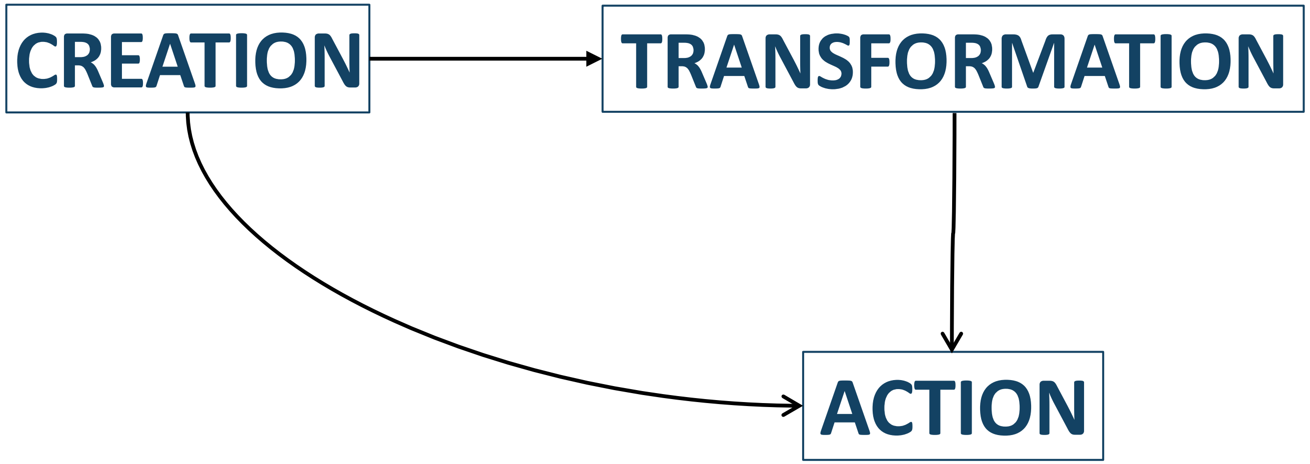
- RDD stands for **Resilient Distributed Datasets.** It is an **immutable** distributed collection of data, which is partitioned across cluster machines
- There are three vital characteristics associated with an RDD:
    - Dependencies
    - Partitions (with some locality information)
    - Compute function: Partition => Iterator[T]

Why they are called RDDs?

- **Resiliency**: a list of dependencies that instructs Spark how an RDD is constructed. When necessary, Spark can recreate an RDD from these dependencies and replicate operations on it (lineage)

- **Distributed: partitions** provide Spark the ability to split the work to parallelize computation on partitions across executors. Locality information reduce the amount of data transmitted over the network
- **Datasets**: because it holds data

The **operations** that can be performed on an RDD can be grouped into three groups:



We will discuss these three types of operations in more detail later, but the main ideas are as follows:

- **Creation:** there are three ways to create an RDD

    1. External source: file or external data sources (kafka, databases, etc) →

    ```
    from pyspark.sql import SparkSession
    spark = (SparkSession.builder.appName("create RDD").getOrCreate())

    newRDD = spark.sparkContext().textFile("data.txt")
    ```

    2. Form an internal structure →

    ```
    data = [1, 2, 3, 4, 5]
    distData = sc.parallelize(data)
    ```

    3. From other RDD, as RDDs are immutable every transformation over one RDD yields another RDD →

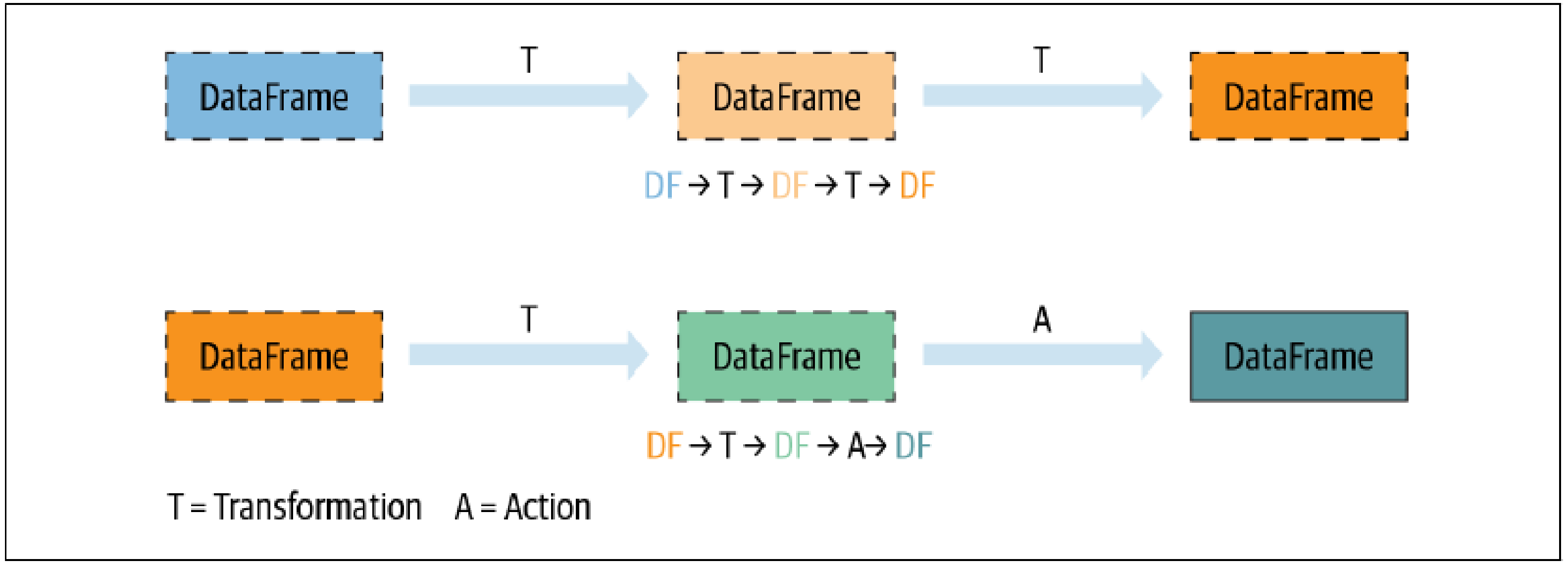
    ```
    newRDD = otherRDD.map(lambda s: (s, 1))
    ```

- **Transformation**
  - All transformations are **evaluated lazily** → Their results are not computed immediately, they are recorded as a **lineage.** This allows Spark to rearrange or optimize certain transformations for **more efficient execution**
  - We can distinguish between narrow and wide transformations

- **Action:** is used to either save a result or to display it. Triggers the execution of all the previous defined transformations

Tranformations vs Actions

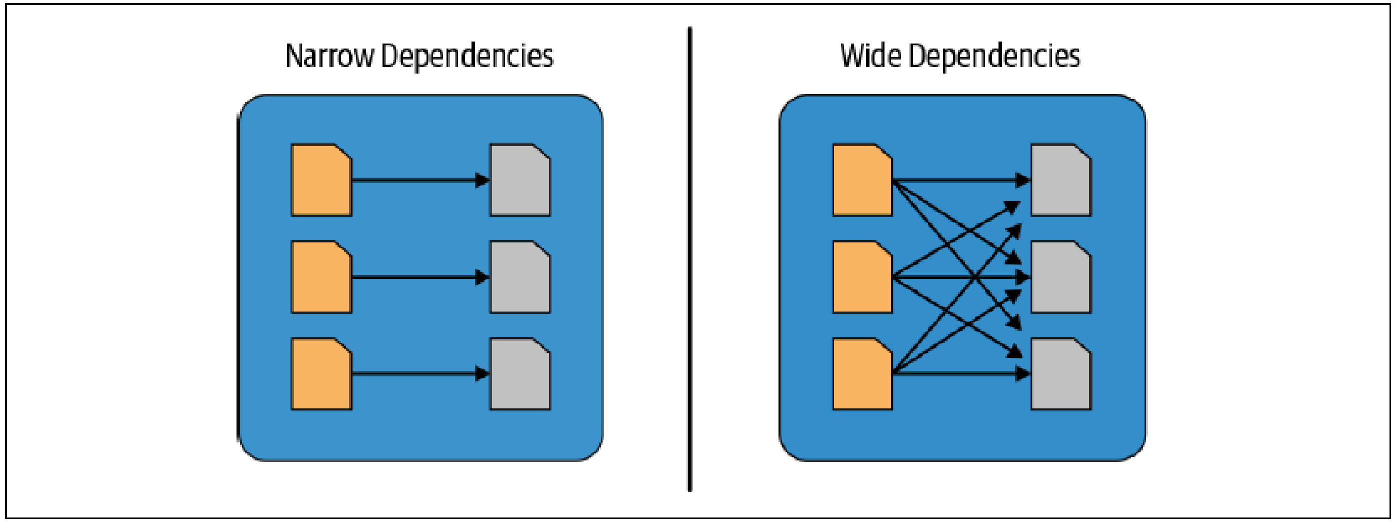
- An action triggers the lazy evaluation of all the recorded transformations
- Each transformation T produces a new DataFrame



Transformations	Actions
orderBy()	show()
groupBy()	take()
filter()	count()
select()	collect()
join()	save()

Narrow vs Wide Transformations

- Transformations can be classified as having either narrow dependencies or wide dependencies
- **Narrow transformation:** when a single output partition can be computed from a single input partition. Ex: filter(), select(), contains()
- **Wide transformation:** when data from other partitions is read in, combined, and written to disk → will force a **shuffle** of data. Ex: orderBy(), groupBy()



Limitations of RDDs

- The compute function and the data type are opaque to Spark, so that:
  - Spark has **no way to optimize** the expression
  - Spark can not use any data compression techniques to serialize the object in the *Iterator[T]*

- The syntax of the compute functions is often cryptic and hard to read
- The syntax will be very different depending on which language and component we use (Python, Scala…)

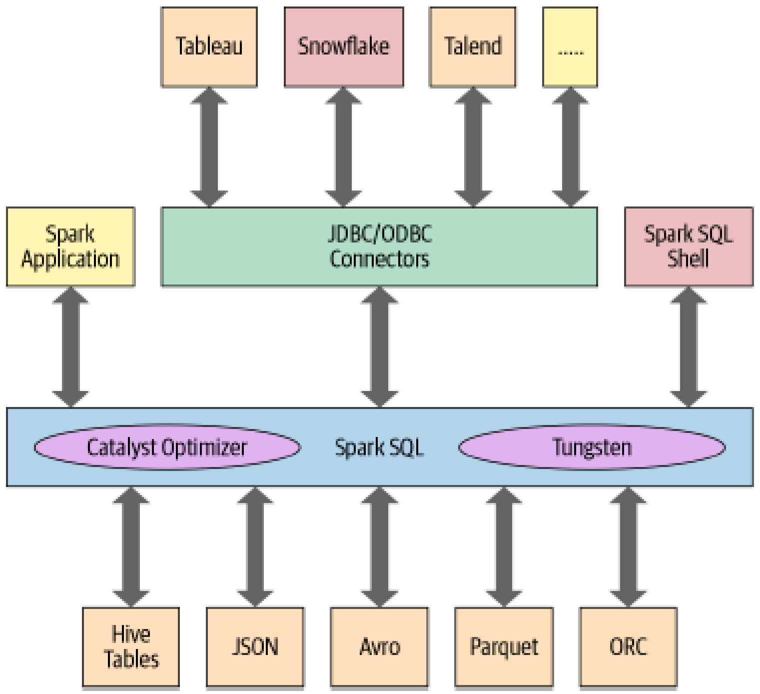
So, in the next section we will see how Spark solves these limitations

2. Spark SQL: DataFrame API

Spark SQL

- So, how can overcome all RDDs limitations? → adding **structure** to Apache Spark & **high-level** expressive operational **functions => SPARK SQL**

- Introduced in Spark 1.3, Spark SQL has evolved into a substantial engine upon which many high-level structured functionalities have been built
- At the core of the Spark SQL engine are the **Catalyst** optimizer and Project Tungsten
- These support the high-level **DataFrame** and **Dataset APIs** and SQL queries



Main Advantages of Spark SQL & High-level APIs

- **Better performance** and space efficiency (Tungsten & Catalyst)
- Expressivity, **simplicity** and composability: the code will be much easier to develop, read, and maintain
- **Uniformity** across its components and languages

We will discuss performance tuning in the later sections, but for now, let's look at some examples of the other advantages

DataFrame API

- Inspired by pandas DataFrames in structure, format, and a few specific operations
- Are like distributed in-memory tables with named columns and **schemas**, where each column has a specific data type
- To a human's eye, a Spark DataFrame is like a table

```
carsDF.show(3)
```

✓ 0.0s

```
+------------+---------+------------+--------------------+
|Acceleration|Cylinders|Displacement|                Name|
+------------+---------+------------+--------------------+
|        12.0|        8|       307.0|chevrolet chevell...|
|        11.5|        8|       350.0|   buick skylark 320|
|        11.0|        8|       318.0|  plymouth satellite|
+------------+---------+------------+--------------------+
only showing top 3 rows
```

```
carsDF.printSchema()
```

✓ 0.0s

```
root
 |-- Acceleration: double (nullable = true)
 |-- Cylinders: long (nullable = true)
 |-- Displacement: double (nullable = true)
 |-- Name: string (nullable = true)
```

In the following example, we want to aggregate all the ages for each name, group by name, and then average the ages. Let's look at the difference between doing it with RDDs or DataFrame API

```
dataRDD = sc.parallelize([
    ("Brooke", 20), ("Denny", 31),
    ("Jules", 30), ("TD", 35),
    ("Brooke", 25)])

agesRDD = (dataRDD
    .map(lambda x: (x[0], (x[1], 1)))
    .reduceByKey(lambda x, y:
        (x[0] + y[0], x[1] + y[1]))
    .map(lambda x: (x[0], x[1][0]/x[1][1])))
```

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import avg

spark = (SparkSession
.builder
.appName("AuthorsAges")
.getOrCreate())

data_df = spark.createDataFrame([
    ("Brooke", 20), ("Denny", 31),
    ("Jules", 30), ("TD", 35),
    ("Brooke", 25)], ["name", "age"])

avg_df = (data_df
    .groupBy("name")
    .agg(avg("age")))
```

SCHEMA

As well as being simpler to read, the structure of Spark's high-level APIs also introduces uniformity across its components and languages. For example, let's compare the logic above with the same logic written in Scala

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import avg

spark = (SparkSession
.builder
.appName("AuthorsAges")
.getOrCreate())

data_df = spark.createDataFrame([
    ("Brooke", 20), ("Denny", 31),
    ("Jules", 30), ("TD", 35),
    ("Brooke", 25)], ["name", "age"])

avg_df = (data_df
    .groupBy("name")
    .agg(avg("age")))
```

```scala
import org.apache.spark.sql.functions.avg
import org.apache.spark.sql.SparkSession

val spark = SparkSession
    .builder
    .appName("AuthorsAges")
    .getOrCreate()

val dataDF = spark.createDataFrame(Seq(
    ("Brooke", 25), ("Denny", 31),
    ("Jules", 30), ("TD", 35),
    ("Brooke", 25))).toDF("name", "age")

val avgDF = dataDF
    .groupBy("name")
    .agg(avg("age"))
```

Spark's Basic Data Types

In the following tables we can see the data types supported by spark (in python)

Table 3-3. Basic Python data types in Spark

Data type	Value assigned in Python	API to instantiate
ByteType	int	DataTypes.ByteType
ShortType	int	DataTypes.ShortType
IntegerType	int	DataTypes.IntegerType
LongType	int	DataTypes.LongType
FloatType	float	DataTypes.FloatType
DoubleType	float	DataTypes.DoubleType
StringType	str	DataTypes.StringType
BooleanType	bool	DataTypes.BooleanType
DecimalType	decimal.Decimal	DecimalType

Table 3-5. Python structured data types in Spark

Data type	Value assigned in Python	API to instantiate
BinaryType	bytearray	BinaryType()
TimestampType	datetime.datetime	TimestampType()
DateType	datetime.date	DateType()
ArrayType	List, tuple, or array	ArrayType(dataType, [nullable])
MapType	dict	MapType(keyType, valueType, [nullable])
StructType	List or tuple	StructType([fields])
StructField	A value type corresponding to the type of this field	StructField(name, dataType, [nullable])

DataFrame Schema

Spark allows you to define a schema in two ways:

- One is to define it programmatically using the Spark DataFrame API

```
from pyspark.sql.types import *

schema = StructType([
    StructField("author", StringType(), False),
    StructField("title", StringType(), False),
    StructField("pages", IntegerType(), False)])
```

- And the other is to employ a Data Definition Language (DDL) string

```
schema = "author STRING, title STRING, pages INT"
```

- When we read from structured files (csv, json, etc) **Spark can** also **infer the Schema** for us. However, is more efficient to define a schema as Spark will not have to go through the DataFrame twice
- Also, Spark can infer schema from a sample at a lesser cost

```
val sampleDF = spark
    .read
    .option("samplingRatio", 0.001)
    .option("header", true)
    .csv("""/databricks-datasets/learning-spark-v2/sf-fire/sf-fire-calls.csv""")
```

- When writing the DataFrame into an external data source Parquet is the default format. The schema is preserved as part of the Parquet metadata. In this case, subsequent reads back into a DataFrame do not require you to manually supply a schema

Rows

- A row in Spark is a generic *Row object*, containing one or more columns
- Each column may be of the same data type, or they can have different types (integer, string, map, array, etc.)
- You can instantiate a Row in each of Spark's supported languages and access its fields by an index starting at 0

```
from pyspark.sql import Row

blog_row = Row(6, "Reynold", 255568, "3/2/2015", ["twitter", "LinkedIn"])

# access using index for individual items
blog_row[1]
'Reynold'
```

- Row objects can be used to create DataFrames if you need them for quick interactivity and exploration

```
rows = [Row("Matei Zaharia", "CA"), Row("Reynold Xin", "CA")]
authors_df = spark.createDataFrame(rows, ["Authors", "State"])
authors_df.show()
```

```
+-------------+-----+
|       Author|State|
+-------------+-----+
|Matei Zaharia|   CA|
|  Reynold Xin|   CA|
+-------------+-----+
```

- In practice, though, you will usually want to read DataFrames from a file or external data source like RDBMS

Now we have explore the basic components of a DataFrame and its advantages, let's take a look in the next section at some of the operations you can perform with DataFrame API

3. Operations With Dataframes

Read and Write

```
# READ
accountDF = (spark.read.option("header", True).option("inferSchema", True)
    .csv("path_value"))

accountDF = (spark.read.option("header", True).option("inferSchema", True)
    .format("csv").load ("path_value"))

# WRITE
accountDF.write.csv("path_value"))

accountDF.write.format("csv").save("path_value")

# we can also save it as a table, against which we can perform queries later
accountDF.write.format("parquet").saveAsTable("table_name")
```

Projections and Filters

```
# PROJECTIONS = select() -> effect on COLUMNS
newDF = accountDF.select("col_name1", "col_name2")

# drop() is the opposite op. of select() -> cols passed as an argument are dropped
newDF = accountDF.drop("col_name")


# FILTERS = filter() or where() -> effect on ROWS
newDF = accountDF.filter(col("col_name") > 800)

newDF = accountDF.filter((col("col_name") > 800) & (col("col_name") < 900))

newDF = accountDF.where(col("col_name") > 800)
```

Projections and Filters Examples

```
accountDF.show()
```

```
+----------+-----------+---------+------+
|first_name|      id_no|available|  debt|
+----------+-----------+---------+------+
|      Rois|909270594-2|   634.92|409.23|
|    Faydra|634513604-2|    10.18|335.53|
|     Edita|296002341-2|   929.67|228.81|
|    Kameko|336828972-1|   833.36|249.93|
|    Rockey|618773088-7|   971.86|383.12|
|     Cecil|758744456-4|    632.9|369.56|
|     Jolee|559569182-4|    43.25|153.27|
|     Gerry|283210905-5|   451.98|277.53|
|    Billye|193221406-2|   994.31|438.67|
|     Ollie|131159756-5|   363.53|377.47|
+----------+-----------+---------+------+
```

```
accountDF2 = accountDF.select("first_name", "available")
accountDF2.show()
```

```
+----------+---------+
|first_name|available|
+----------+---------+
|      Rois|   634.92|
|    Faydra|    10.18|
|     Edita|   929.67|
|    Kameko|   833.36|
|    Rockey|   971.86|
|     Cecil|    632.9|
|     Jolee|    43.25|
|     Gerry|   451.98|
|    Billye|   994.31|
|     Ollie|   363.53|
+----------+---------+
```

```
accountDF3 = accountDF.filter((col("available") > 800)
                                  & (col("debt") < 300))

accountDF3.show()
```

```
+----------+-----------+---------+------+
|first_name|      id_no|available|  debt|
+----------+-----------+---------+------+
|     Edita|296002341-2|   929.67|228.81|
|    Kameko|336828972-1|   833.36|249.93|
+----------+-----------+---------+------+
```

Renaming & Adding Columns

```
# RENAMING COLS
newDF = accountDF.withColumnRenamed("existing_col", "new_name")


# ADDING COLS
newDF = accountDF.withColumn("new_col_name", col("existing_col"))

# we can perform opperations with more than one col
newDF = accountDF.withColumn("new_col_name ", col("col_name1")-col("col_name2"))

# or assign constant value to this col with lit()
newDF = accountDF.withColumn("new_col_name", lit("value"))
```

Renaming & Adding Columns Examples

```
accountDF.show()
```

```
+----------+-----------+---------+------+
|first_name|      id_no|available|  debt|
+----------+-----------+---------+------+
|      Rois|909270594-2|   634.92|409.23|
|    Faydra|634513604-2|    10.18|335.53|
|     Edita|296002341-2|   929.67|228.81|
|    Kameko|336828972-1|   833.36|249.93|
|    Rockey|618773088-7|   971.86|383.12|
|     Cecil|758744456-4|    632.9|369.56|
|     Jolee|559569182-4|    43.25|153.27|
|     Gerry|283210905-5|   451.98|277.53|
|    Billye|193221406-2|   994.31|438.67|
|    Ollie|131159756-5|   363.53|377.47|
+----------+-----------+---------+------+
```

```
accountDF4 = accountDF.withColumnRenamed("available", "avbl")
accountDF4.show()
```

```
+----------+-----------+------+------+
|first_name|      id_no|  avbl|  debt|
+----------+-----------+------+------+
|      Rois|909270594-2|634.92|409.23|
|    Faydra|634513604-2| 10.18|335.53|
|     Edita|296002341-2|929.67|228.81|
|    Kameko|336828972-1|833.36|249.93|
|    Rockey|618773088-7|971.86|383.12|
|     Cecil|758744456-4| 632.9|369.56|
|     Jolee|559569182-4| 43.25|153.27|
|     Gerry|283210905-5|451.98|277.53|
|    Billye|193221406-2|994.31|438.67|
|     Ollie|131159756-5|363.53|377.47|
+----------+-----------+------+------+
```

```
+----------+-----------+---------+------+-------+
|first_name|      id_no|available|  debt|balance|
+----------+-----------+---------+------+-------+
|      Rois|909270594-2|   634.92|409.23| 225.69|
|    Faydra|634513604-2|    10.18|335.53|-325.35|
|     Edita|296002341-2|   929.67|228.81| 700.86|
|    Kameko|336828972-1|   833.36|249.93| 583.43|
|    Rockey|618773088-7|   971.86|383.12| 588.74|
|     Cecil|758744456-4|    632.9|369.56| 263.34|
|     Jolee|559569182-4|    43.25|153.27|-110.02|
|     Gerry|283210905-5|   451.98|277.53| 174.45|
|    Billye|193221406-2|   994.31|438.67| 555.64|
|     Ollie|131159756-5|   363.53|377.47| -13.94|
+----------+-----------+---------+------+-------+
```

```
accountDF5 = accountDF.withColumn("balance", round(col("available")-col("debt"), 2))
accountDF5.show()
```

Hands-on

Open the following python notebooks. Execute the examples and solve the proposed exercises:

- *01.DataFramesBasics*
- *02.ColumnsAndExpressions*

Aggregations

- This type of transformations will have an effect on both **columns** and **rows**
- Almost all these transformations will involve **shuffling** (wide transformations)
- We can distinguish two "steps", the aggregation **CRITERIA**, and the aggregation **OPERATION**

```
newDF = productDF.groupBy("category").agg(sum("kg"))
```

$\underbrace{\texttt{.groupBy("category")}}_{\text{CRITERIA}}\ \underbrace{\texttt{.agg(sum("kg"))}}_{\text{OPERATION}}$

So, in this case, we will get one record for each different product ***category***, so this is the "rule" we are grouping with

On the other hand, for each of these records we will have the sum of the ***kg*** column values for all the records that have the same category

Aggregations Examples

```
productDF.show()
```

```
+--------------------------+-----------+------+------+
|product                   |category   |origin|kg    |
+--------------------------+-----------+------+------+
|Crab Meat Claw Pasteurise |Frozen     |France|334.23|
|Coffee - Decaffeinato Coffee|Commodities|Spain |191.84|
|Crab - Claws, 26 - 30     |Frozen     |Italy |331.52|
|Sauce - Marinara          |Groceries  |Spain |211.87|
|Beef - Ground Medium      |Fresh      |France|334.83|
|Truffle Cups - Red        |Groceries  |Italy |137.27|
|Pie Filling - Apple       |Groceries  |France|88.18 |
|Mop Head - Cotton, 24 Oz  |Commodities|Italy |326.82|
|Lamb - Whole, Fresh       |Fresh      |France|294.87|
|Tea - Decaf 1 Cup         |Commodities|France|125.15|
+--------------------------+-----------+------+------+
```

```
productDF2 = (productDF
    .groupBy("category")
    .agg(sum("kg")))

productDF2.show()
```

```
+-----------+-------+
|   category|sum(kg)|
+-----------+-------+
|  Groceries| 437.32|
|Commodities| 643.81|
|      Fresh|  629.7|
|     Frozen| 665.75|
+-----------+-------+
```

```
productDF3 = (productDF
    .groupBy("origin")
    .agg(sum("kg").alias("total_kg")))

productDF3.show()
```

```
+------+--------+
|origin|total_kg|
+------+--------+
|France| 1177.26|
| Italy|  795.61|
| Spain|  403.71|
+------+--------+
```

Aggregations

- As we have seen in the previous example, aggregations can change both the number of rows and the number of columns (although this does not always have to be the case)

- IMPORTANT: the *.groupBy* operation does NOT return another DataFrame by itself, we need to apply some grouping operation (*.agg*)

- Other common aggregation operations for which we have a simplified syntax are: *min(), max(), sum(), avg(), count()*, etc. For example:

```
newDF = productDF.groupBy("category").min("kg")

newDF = productDF.groupBy("category").avg("kg")
```

Joins

- Another common DataFrame operation is to join **two DataFrames** together
- As a result, we will have a new DF with the information of both original DFs
- To perform this operation we have to provide a **joining condition**, which will be one (or more) equality between two columns, one from each of the DFs

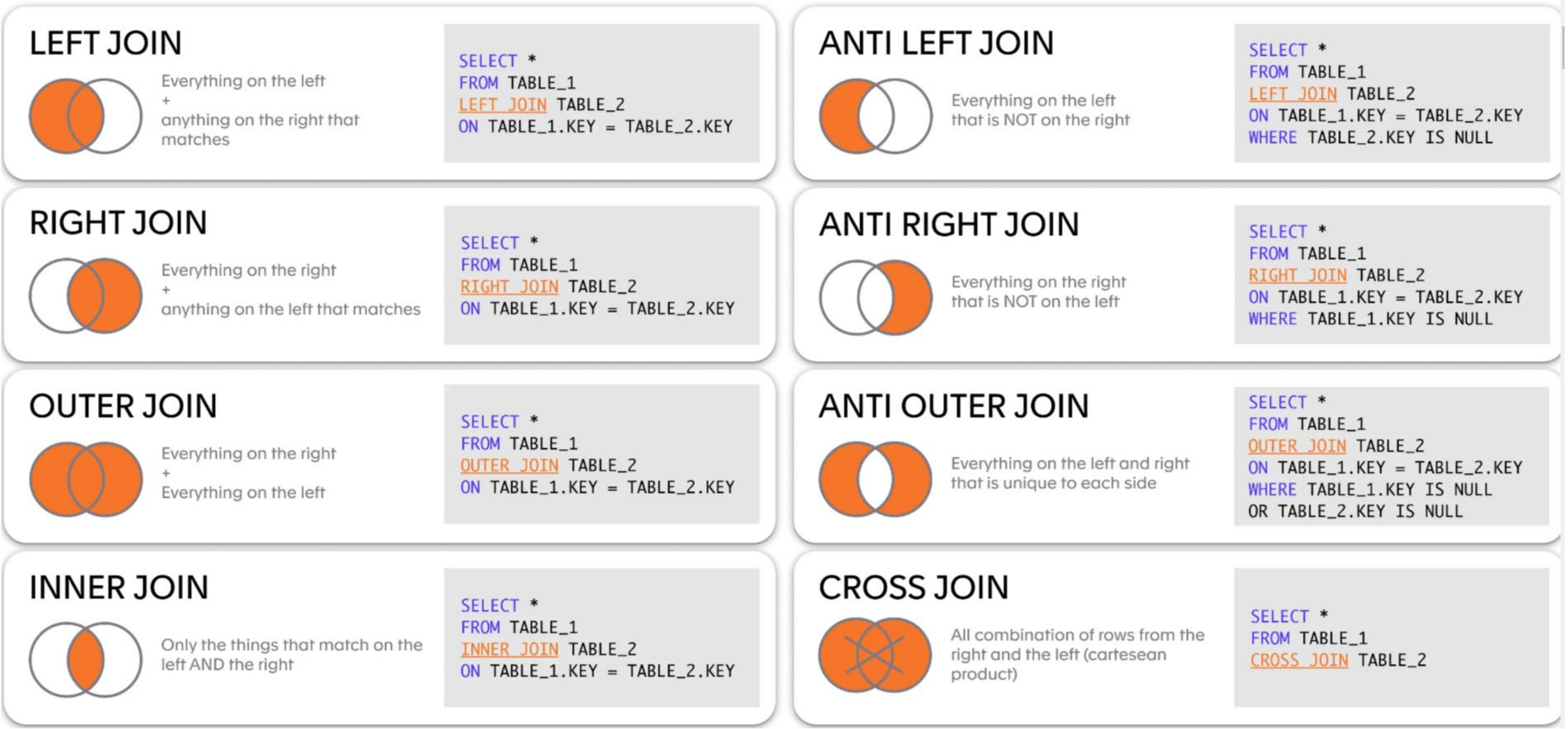
```
joined_df = employee_df.join(orders_df, col('col_name') == col('col_name'), 'inner')
```

DF1 **DF2** **DF1 column** **DF2 column**

JOINING CONDITION **JOIN TYPE**

- We have to provide also the **join type**, we will discuss the different join types in the following slides

Types of Joins



Joins Examples

```
+------+-----+
|letter|value|
+------+-----+
|     A|    1|
|     B|    2|
|     C|    3|
|     D|    4|
+------+-----+
```

```
+------+------+
|letter|number|
+------+------+
|     X|     5|
|     B|     6|
|     C|     7|
|     Z|     8|
+------+------+
```

```
joined_df1 = df1.join(df2, 'letter', 'inner')
joined_df1.show()
```

```
+------+-----+------+
|letter|value|number|
+------+-----+------+
|     B|    2|     6|
|     C|    3|     7|
+------+-----+------+
```

```
joined_df2 = df1.join(df2, 'letter', 'left')
joined_df2.show()
```

```
+------+-----+------+
|letter|value|number|
+------+-----+------+
|     A|    1|  null|
|     B|    2|     6|
|     C|    3|     7|
|     D|    4|  null|
+------+-----+------+
```

```
joined_df3 = df1.join(df2, 'letter', 'right')
joined_df3.show()
```

```
+------+-----+------+
|letter|value|number|
+------+-----+------+
|     X| null|     5|
|     B|    2|     6|
|     C|    3|     7|
|     Z| null|     8|
+------+-----+------+
```

Joins Examples

```
+------+-----+
|letter|value|
+------+-----+
|     A|    1|
|     B|    2|
|     C|    3|
|     D|    4|
+------+-----+

+------+------+
|letter|number|
+------+------+
|     X|     5|
|     B|     6|
|     C|     7|
|     Z|     8|
+------+------+
```

```
joined_df4 = df1.crossJoin(df2)
joined_df4.show()
```

```
+------+-----+------+------+
|letter|value|letter|number|
+------+-----+------+------+
|     A|    1|     X|     5|
|     B|    2|     X|     5|
|     C|    3|     X|     5|
|     D|    4|     X|     5|
|     A|    1|     B|     6|
|     B|    2|     B|     6|
|     C|    3|     B|     6|
|     D|    4|     B|     6|
|     A|    1|     C|     7|
|     B|    2|     C|     7|
|     C|    3|     C|     7|
|     D|    4|     C|     7|
|     A|    1|     Z|     8|
|     B|    2|     Z|     8|
|     C|    3|     Z|     8|
|     D|    4|     Z|     8|
+------+-----+------+------+
```

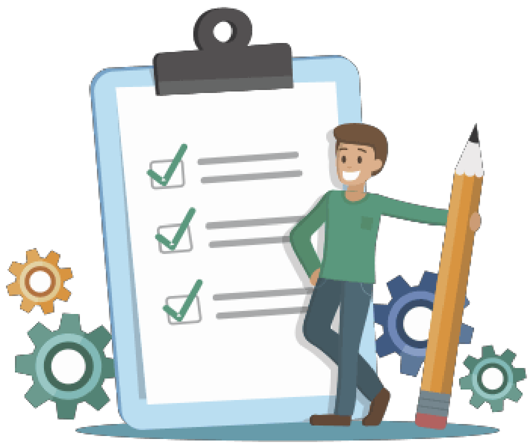
```
joined_df5 = df1.join(df2, 'letter', 'outer')
joined_df5.show()
```

```
+------+-----+------+
|letter|value|number|
+------+-----+------+
|     A|    1|  null|
|     B|    2|     6|
|     C|    3|     7|
|     D|    4|  null|
|     X| null|     5|
|     Z| null|     8|
+------+-----+------+
```

Hands-on

Open the following python notebooks. Execute the examples and solve the proposed exercises:

- *03.Aggregations*
- *04.Joins*



4. Advanced Transformations

Window Partitioning

- A window function uses values from the rows in a window (a range of input rows) to return a set of values
- Is like creating a DF for each different data column (or columns) of the partitionBy(), and applying the specified operation

```
windowDF = df3.withColumn("order",
    row_number().over(Window.partitionBy("col_partition").orderBy("col_name")))
```

operation

partition col

- We have to use withColumn to store the results in a new col
- There are different operations available like rank(), dense_rank(), first_value(), etc
- The operations that implies ordering needs a column (or columns) as an ordering criteria

Windowing Examples

```
+------+-----+-----+
|letter|value|order|
+------+-----+-----+
|     A|   10|    1|
|     A|   20|    2|
|     B|   30|    1|
|     B|   40|    2|
|     C|   10|    1|
+------+-----+-----+
```

```
windowDF = df3.withColumn("order",
    row_number()
    .over(Window.partitionBy("letter")
    .orderBy("value")))

windowDF.show()
```

```
+------+-----+-----+
|letter|value|order|
+------+-----+-----+
|     A|   10|    1|
|     A|   20|    2|
|     B|   30|    1|
|     B|   40|    2|
|     C|   10|    1|
+------+-----+-----+
```

```
windowDF2 = df3.withColumn("order",
    min("value")
    .over(Window.partitionBy("letter")))

windowDF2.show()
```

```
+------+-----+-----+
|letter|value|order|
+------+-----+-----+
|     A|   20|   10|
|     A|   10|   10|
|     B|   30|   30|
|     B|   40|   30|
|     C|   10|   10|
+------+-----+-----+
```

UDFs

We can also define our own functions

```
+------+-----+
|letter|value|
+------+-----+
|  ABCD|    1|
|  BCDE|    2|
|  CDEF|    3|
|  DEFG|    4|
+------+-----+
```

```
twoChar = udf(lambda s: s[:2])

twoDF = df4.withColumn("two", twoChar(col("letter")))
twoDF.show()
```

```
+------+-----+---+
|letter|value|two|
+------+-----+---+
|  ABCD|    1| AB|
|  BCDE|    2| BC|
|  CDEF|    3| CD|
|  DEFG|    4| DE|
+------+-----+---+
```

```
from pyspark.sql.types import IntegerType

@udf(returnType=IntegerType())
def add_one(x):
    if x is not None:
        return x + 1

plus1DF = df4.withColumn("plus1", add_one(col("value")))
plus1DF.show()
```

```
+------+-----+-----+
|letter|value|plus1|
+------+-----+-----+
|  ABCD|    1|    2|
|  BCDE|    2|    3|
|  CDEF|    3|    4|
|  DEFG|    4|    5|
+------+-----+-----+
```

- However, it is preferable whenever possible to use Spark's built-in transformations, especially when working with Python

Hands-on

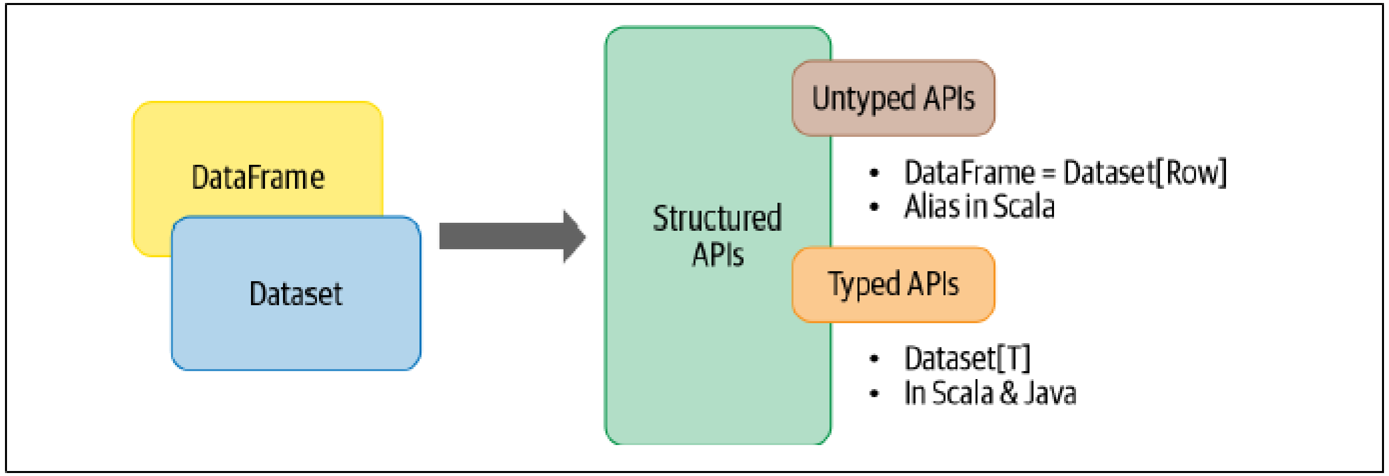
Open the following python notebooks. Execute the examples and solve the proposed exercises:

- *05.WindowPartitioning_UDFs*

5. DataSet API

DataSet API

- A Dataset is a collection of strongly typed JVM objects in Scala or a class in Java
- In Spark’s supported languages, Datasets make sense only in Java and Scala
- In Python and R only DataFrames make sense because Python and R are not compile-time type-safe. Types are dynamically inferred or assigned during execution
- Conceptually, you can think of a DataFrame in Scala as an alias for a collection of generic objects, Dataset[Row],



Creating DataSets

- When creating a Dataset in Scala, the easiest way to specify the schema for the resulting Dataset is to use a case class
- In Java, JavaBean classes are used
- Let's take a look at this example, a reading from an IoT device in a JSON file
- To express each JSON entry as DeviceIoTData, a domain-specific object, we can define a Scala case class


```
{"device_id": 198164, "device_name": "sensor-pad-198164owomcJZ", "ip":
"80.55.20.25", "cca2": "PL", "cca3": "POL", "cn": "Poland", "latitude":
53.080000, "longitude": 18.620000, "scale": "Celsius", "temp": 21,
"humidity": 65, "battery_level": 8, "c02_level": 1408,"lcd": "red",
"timestamp" :1458081226051}
```


```scala
case class DeviceIoTData (battery_level: Long, c02_level: Long,
cca2: String, cca3: String, cn: String, device_id: Long,
device_name: String, humidity: Long, ip: String, latitude: Double,
lcd: String, longitude: Double, scale:String, temp: Long,
timestamp: Long)
```

Creating DataSets

- Once defined, we can use it to read our file and convert the returned Dataset[Row] into Dataset[DeviceIoTData]

```
case class DeviceIoTData (battery_level: Long, c02_level: Long,
cca2: String, cca3: String, cn: String, device_id: Long,
device_name: String, humidity: Long, ip: String, latitude: Double,
lcd: String, longitude: Double, scale:String, temp: Long,
timestamp: Long)
```

```
// In Scala
val ds = spark.read
    .json("/path/example.json")
    .as[DeviceIoTData]
```

- Although with JSON and CSV data it’s possible to infer the schema, for large data sets this is resource-intensive (expensive)
- Just as with DataFrames, you can perform transformations and actions on DataFrames. Let´s look at an example

Transformations and Actions with DataSets

- Once defined, we can use it to read our file and convert the returned Dataset[Row] into Dataset[DeviceIoTData]

```
// In Scala
val filterTempDS = ds.filter({d => {d.temp > 30 && d.humidity > 70}})
```

- A thing to note is that, while with DataFrames you express your filter() conditions as SQL-like DSL operations which are language-agnostic, with Datasets we use language-native expressions as Scala or Java code

DataFrames vs Datasets

So, when it is preferable to use DataSets rather than Dataframes? In many cases either will work, but there are some situations where one is preferable to the other. Here are a few examples:

- If you want strict compile-time type safety and don't mind creating multiple case classes for a specific Dataset[T], use Datasets
- If you want to take advantage of and benefit from Tungsten's efficient serialization with Encoders, use Datasets
- If your processing dictates relational transformations similar to SQL-like queries, use DataFrames
- If you want unification, code optimization, and simplification of APIs across Spark components, use DataFrames

DataFrames vs Datasets

- If you are an R user, use DataFrames.
- If you are a Python user, use DataFrames and drop down to RDDs if you need more control.
- If you want space and speed efficiency, use DataFrames
- If you want errors caught during compilation rather than at runtime, choose the appropriate API as depicted in figure below

Structured APIs In Spark

	SQL	DataFrames	Datasets
Syntax Errors	Runtime	Compile Time	Compile Time
Analysis Errors	Runtime	Runtime	Compile Time

RDDs vs Structured APIs

In addition, there are some scenarios where you will want to consider using RDDs, such as when you:

- Are using a third-party package that's written using RDDs
- Can forgo the code optimization, efficient space utilization, and performance benefits available with DataFrames and Datasets
- Want to precisely instruct Spark *how to do* a query

But use DataFrames or DataSets when:

- You want to tell Spark *what to do*, not *how to do* it
- If you want rich semantics, high-level abstractions, and DSL operators
- If your processing demands high-level expressions, SQL queries, columnar access, or use of relational operators on semi-structured data