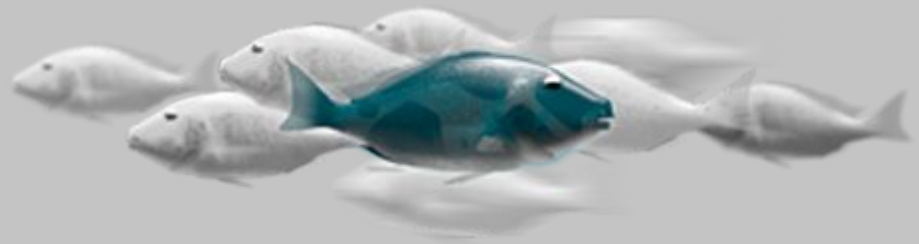
EDEM



Máster en Big Data & Cloud | VII Edición

Pyspark

Ignacio Reyes Vázquez

1. Estructuras de datos de Apache Spark: RDD

Resilient Distributed Datasets (RDD)

- RDD son las siglas de Resilient Distributed Datasets (conjuntos de datos distribuidos y resilientes). Un RDD es una colección de datos repartida entre varias máquinas del clúster y que, además, es inmutable.

- Hay tres características fundamentales asociadas a un RDD:

    1. Particiones (con cierta información de localidad).
    2. Función de cálculo: Partición => Iterador[T].
    3. Dependencias.

```
[1, 2, 3, 4, 5, 6]
```

```
Partición 1: [1, 2, 3]
Partición 2: [4, 5, 6]
```

```
Partición 1 → [1, 2, 3] → función: x*2 → [2, 4, 6]
Partición 2 → [4, 5, 6] → función: x*2 → [8, 10, 12]
```

```
RDD resultante: [2, 4, 6, 8, 10, 12]
Depende del RDD original [1, 2, 3, 4, 5, 6] y de la transformación "multiplicar por 2".
Si una partición se pierde, Spark puede reconstruirla usando esta dependencia.
Esto es la resiliencia: no hace falta otra copia, Spark la regenera.
```

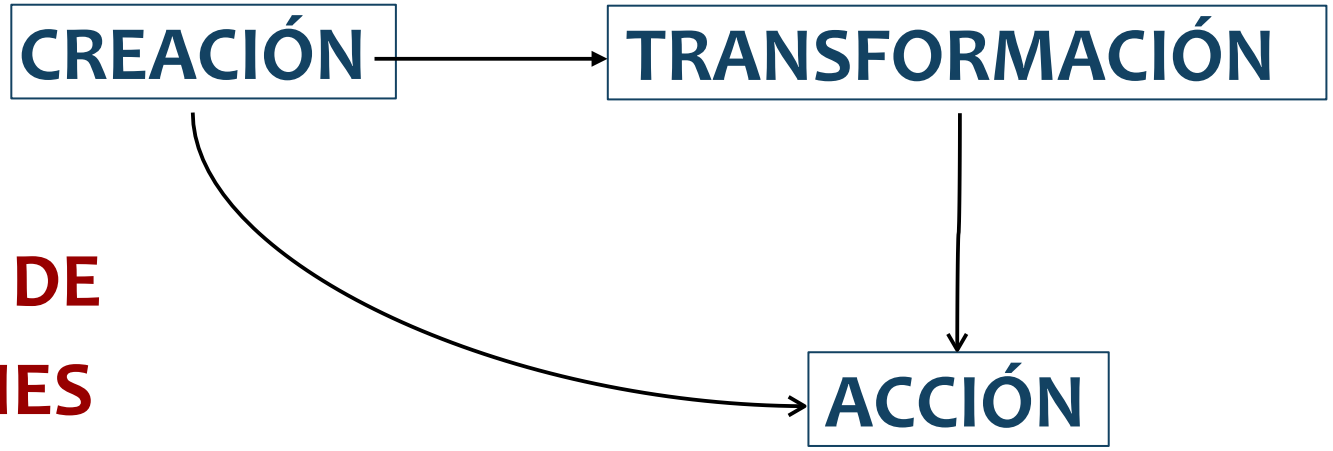
¿Por qué se llaman RDD?

- Resilient (Resiliencia): conserva una lista de dependencias que indica a Spark cómo fue generado. Gracias a esta información, Spark puede volver a crear el RDD cuando sea necesario y repetir las operaciones que lo formaron (linaje).

- Distributed (Distribuidas): las particiones proporcionan a Spark la capacidad de dividir el trabajo para paralelizar el cálculo en particiones entre ejecutores. La información de localidad reduce la cantidad de datos transmitidos a través de la red.

- Datasets: porque contiene datos.

TRES TIPOS DE OPERACIONES



Tipos de Operaciones con Spark

- **Creación:** Hay tres formas diferentes de crear una RDD

1. Fuente externa: archivo o fuentes de datos externas (kafka, bases de datos, etc.) →

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

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

2. Desde una estructura interna →

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

3. Desde otro RDD, ya que los RDD son inmutables, cada transformación sobre un RDD produce otro → `newRDD = dataRDD.map(lambda s: (s, 1))`

Tipos de Operaciones con Spark

- Las **Transformaciones** son operaciones diferidas (Lazy) que crean un nuevo RDD. Solo registran la intención de la operación (el linaje) para permitir que Spark optimice el plan de ejecución completo, sin calcular nada inmediatamente.

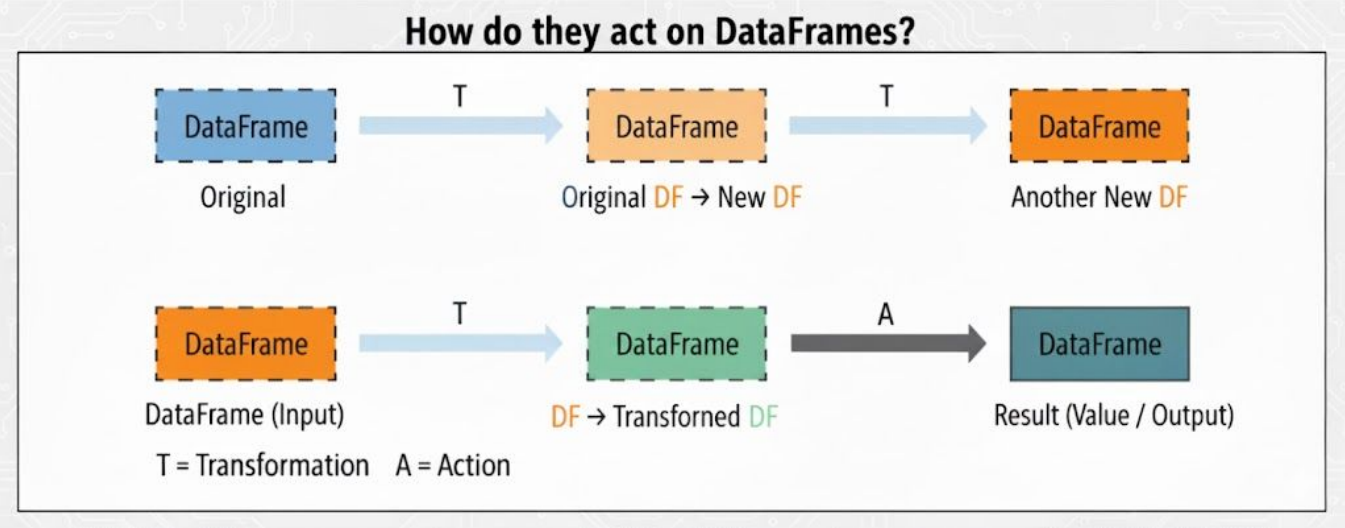
- Las **Acciones** son la operación final que fuerza la ejecución de todas las transformaciones registradas. Devuelven un resultado concreto al driver o lo escriben en un sistema de almacenamiento.

```
# TRANSFORMACIÓN (Diferida): Registra la intención de filtrar, NO se ejecuta el
rdd_filtrado = rdd_original.filter(lambda x: x > 5)

# ACCIÓN (Inmediata): Fuerza la ejecución del 'filter' y devuelve el resultado.
total = rdd_filtrado.count()
```

Transformaciones vs Acciones

- Una **Acción** es el 'Botón de Inicio' que obliga a Spark a ejecutar la evaluación de todas las Transformaciones registradas previamente en el **linaje.**
- Cada **Transformación** crea siempre un DataFrame totalmente nuevo (**Inmutabilidad**), asegurando que el DataFrame original permanezca intacto y permitiendo la recuperación de datos en caso de fallos.



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

Tipos de Transformaciones

- Una transformación es **narrow** (estrecha) cuando la información necesaria para calcular una nueva partición se encuentra completamente en una única partición de entrada del RDD o DataFrame padre. Ej: .filter(), .select(), .map()
- Una transformación es **wide** (amplia) cuando la información necesaria para calcular una única partición de salida proviene de varias (o todas) las particiones de entrada (genera un *SHUFFLE*). Ej: .groupBy(), .orderBy(), .join()

Partición	Categoría	Ventas
P1	Electrónica, Ropa	100, 50
P2	Ropa, Electrónica	30, 80
P3	Alimentos, Alimentos	20, 10

```
rpp_df = productos_df.filter(col("Ventas") > 40)
```

Queremos obtener solo las filas donde las ventas son mayores a 40.

```
ventas_totales_df = productos_df.groupBy("Categoria").sum("Ventas")
```

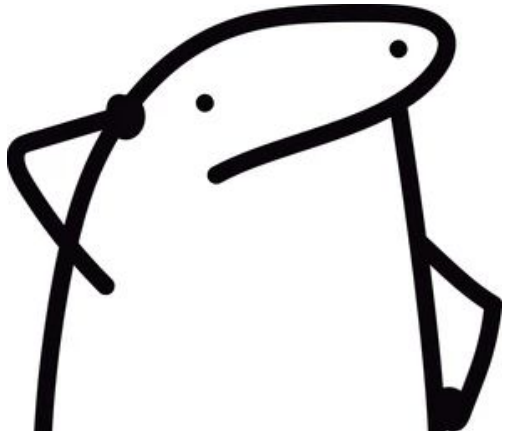
Queremos calcular la suma total de `Ventas` para cada `Categoria`.

Limitaciones de las RDDs

La función de cálculo y el tipo de datos son opacos para Spark, por lo que:

- Spark no puede optimizar la expresión ni utilizar técnicas de compresión de datos al serializar los objetos (Iterator[T]), lo que impacta el rendimiento.

- La sintaxis de las funciones (lambda) suele ser críptica y difícil de leer, variando mucho según el lenguaje (Python, Scala).



¿Spark tiene alguna opción para superar estas limitaciones?

2. Spark SQL: DataFrame API

Spark SQL

1. **Añadiendo Estructura:** Se superó la opacidad de los RDDs al introducir estructura (esquemas) a los datos, dando origen a las API de alto nivel: DataFrame y Dataset.
2. **Motor Central**: Spark SQL se convirtió en el motor fundamental de Spark (introducido en la v1.3), soportando tanto consultas SQL como estas nuevas API estructuradas.
3. **Optimizador Catalyst:** Este componente clave analiza, planifica y optimiza las consultas estructuradas, eliminando la pérdida de rendimiento que ocurría con el código opaco.
4. **Proyecto Tungsten:** Se enfocó en la eficiencia a nivel de hardware, mejorando la serialización de datos y la gestión de memoria para acelerar la transferencia de datos entre workers.

DataFrame API

- Inspirado en los **pandas DataFrames** en cuanto a estructura, formato y algunas operaciones específicas (permite el uso de **Python**, Scala, R o Java).
- Son como tablas distribuidas en memoria con **columnas y esquemas** con nombre, donde cada columna tiene un tipo de datos específico.
- A simple vista, un Spark DataFrame es como una **tabla**.

```
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)
```

Diferencia entre el uso de RDDs y DFs

```
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

Estructura común en el uso de APIs

PYSPARK

```
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")))
```

SPARK 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"))
```

Tipos de datos soportados por Pyspark

Tipo de Dato Spark	Equivalente en Python	Clase API para Instanciar	Categoría
StringType	str	DataTypes.StringType	Básico
IntegerType	int	DataTypes.IntegerType	Básico
LongType	int	DataTypes.LongType	Básico
DoubleType	float	DataTypes.DoubleType	Básico
BooleanType	bool	DataTypes.BooleanType	Básico
DateType	datetime.date	DateType()	Estructurado
TimestampType	datetime.datetime	TimestampType()	Estructurado
ArrayType	list, tuple o array	ArrayType(dataType, [nullable])	Estructurado
MapType	dict	MapType(keyType, valueType, [nullable])	Estructurado
StructType	List o tuple	StructType([fields])	Estructurado

DataFrame Schema

Spark te permite definir un esquema de dos maneras:

- Definirlo mediante programación utilizando la API Spark DataFrame

```
from pyspark.sql.types import *

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

- Utilizar una cadena de lenguaje de definición de datos (DDL), utilizada en SQL

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

```
books_df = spark.createDataFrame(data, schema=schema)
```

- Spark es capaz de inferir un esquema cuando estás leyendo por ejemplo un csv, pero no es recomendado
- Al guardar el archivo, el esquema también es guardado

Objetos Row

- Una fila en Spark es un objeto Row genérico que contiene una o más columnas.
- Cada columna puede ser del mismo tipo de datos o pueden tener tipos diferentes (entero, cadena, mapa, matriz, etc.).
- Puede instanciar una fila en cada uno de los lenguajes compatibles con Spark y acceder a sus campos mediante un índice que comienza en 0.
- Los objetos Row se pueden utilizar para crear DataFrames si los necesita para una interactividad y exploración rápidas (normalmente usaremos .parquet, .csv …)

```
from pyspark.sql import Row

blog_row = Row(6, "Reynold", 255568, "3/2/2015", ["twitter", "LinkedIn"])
blog_row[1] 'Reynold'

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

3. Operaciones con Dataframes

Leer y Escribir

```
# Leer
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"))

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

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

# Podemos guardarlo en formato tabla para posteriormente utilizar queries para consultarla
accountDF.write.format("parquet").saveAsTable("table_name")
```

Proyecciones y Filtros

```
# Proyecciones = select() -> afecta a las COLUMNAS
newDF = accountDF.select("col_name 1", "col_name 2")

# drop() es el opuesto del select() -> las columnas pasadas como argumento se eliminan
newDF = accountDF.drop("col_name")


# Filtros = filter() o where() -> afecta a FILAS
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)
```

Ejemplo de Proyecciones y Filtros

accountDF.show()

```
+----------+---------+-------+-----+
|first_name|available|   debt| cost|
+----------+---------+-------+-----+
|     Alice|     true| 1500.5|45.99|
|       Bob|    false|    0.0|120.0|
|   Charlie|     true| 890.75| 10.5|
|     David|    false| 3200.0|200.0|
|       Eve|     true|   50.0|15.25|
|     Frank|    false|12000.0|300.0|
|     Grace|     true| 450.99| 5.99|
|     Heidi|     true|  750.0| 88.5|
|      Ivan|    false|  900.0| 60.0|
|      Judy|     true|  100.0| 12.5|
+----------+---------+-------+-----+
```

```
accountDF3 = accountDF.filter((accountDF["available"]) & (accountDF["debt"] < 300))
accountDF3.show()
```

```
+----------+---------+-----+-----+
|first_name|available| debt| cost|
+----------+---------+-----+-----+
|       Eve|     true| 50.0|15.25|
|      Judy|     true|100.0| 12.5|
+----------+---------+-----+-----+
```

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

```
+----------+---------+
|first_name|available|
+----------+---------+
|     Alice|     true|
|       Bob|    false|
|   Charlie|     true|
|     David|    false|
|       Eve|     true|
|     Frank|    false|
|     Grace|     true|
|     Heidi|     true|
|      Ivan|    false|
|      Judy|     true|
+----------+---------+
```

Renombrar y Añadir Columnas

```
# Renombrar Columnas
newDF = accountDF.withColumnRenamed("existing_col", "new_name")


# Añadir Columnas
newDF = accountDF.withColumn("new_col_name", col("existing_col"))

# Podemos hacer operaciones con más de una columna a la vez
newDF = accountDF.withColumn("new_col_name ", col("col_name1")-col("col_name2"))

# Podemos también establecer el valor constante con lit() a una columna
newDF = accountDF.withColumn("new_col_name", lit("value"))
```

Ejemplo de Renombrar y Añadir Columnas

```
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()
```

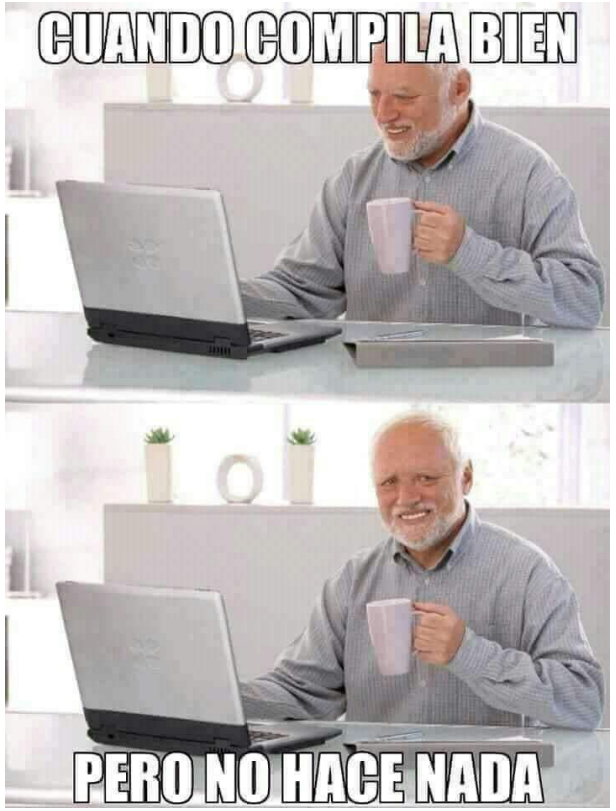
```
+----------+-----------+------+------+
|first_name|      id_no|  avbl|  debt|
+----------+-----------+------+------+
|      Rois|909270594-2|634.92|409.23|
|    Faydra|634513604-2| 10.18|335.53|
|     Edita|296002341-2|929.67|228.81|
|    Kameko|336828972-1|833.36|249.93|
|    Rockey|618773088-7|971.86|383.12|
|     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()
```

EJERCICIOS

- 01.DataFramesSpark
- 02.ExpressionsSpark



Agregaciones

- Este tipo de transformaciones afectarán tanto a las columnas como a las filas.
- Casi todas estas transformaciones implican un SHUFFLE (transformaciones WIDE).
- Podemos distinguir dos «pasos»: los CRITERIOS de agregación y la OPERACIÓN de agregación.

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

CRITERIO OPERACIÓN

Por lo tanto, en este caso, obtendremos un registro por cada categoría de producto diferente, por lo que esta es la «regla/criterio» con la que estamos agrupando.

Por otro lado, para cada uno de estos registros tendremos la suma de los valores de la columna kg para todos los registros que tengan la misma categoría.

Ejemplo de Agregaciones

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|
+------+--------+
```

groupBy en Agregaciones

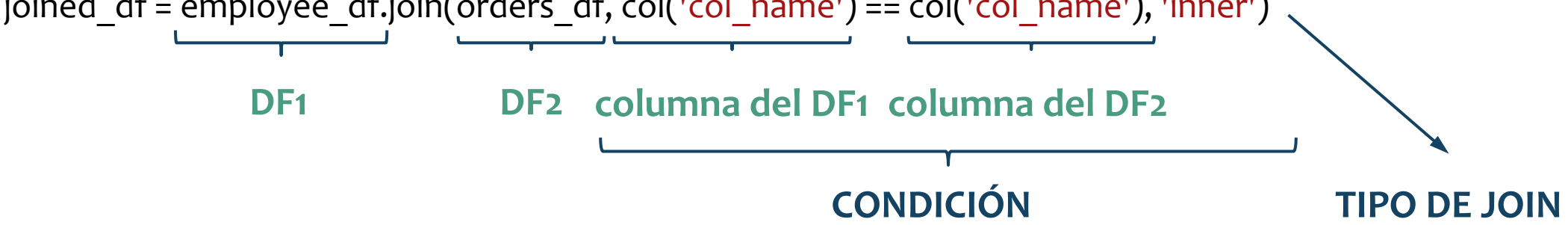
- Las transformaciones de agrupación (como .groupBy()) requieren una función de agregación (.sum(), .avg(), .count()) inmediatamente después para colapsar las filas en un único valor.

- Las transformaciones de reordenación o combinación (como .orderBy() o .join()) son amplias porque fuerzan un Shuffle para mover datos, pero su resultado ya es un nuevo DataFrame (reorganizado o unido) sin necesidad de colapsar filas.

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

Joins

- Otra operación común con DataFrames es unir dos de ellos.
- Como resultado, obtendremos un nuevo DF con la información de ambos DF originales.
- Para realizar esta operación, debemos proporcionar una condición de unión, que será una (o más) igualdad entre dos columnas, una de cada uno de los DF.



Tipos de Join

Ejemplo de Joins

```
+------+-----+
|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|
+------+-----+------+
```

Ejemplo de Joins

```
+------+-----+
|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()
```

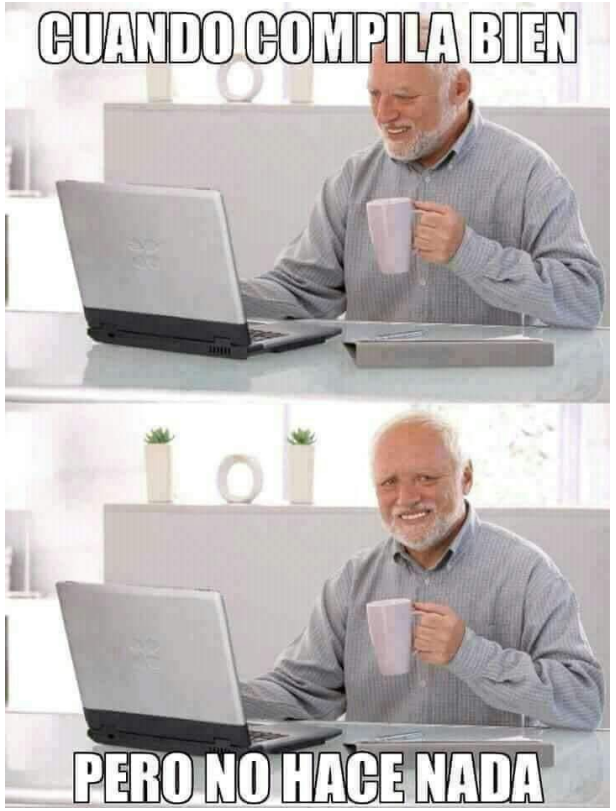
```
+------+-----+------+------+
|letter|value|letter|number|
+------+-----+------+------+
|     A|    1|     X|     5|
|     B|    2|     X|     5|
|     C|    3|     X|     5|
|     D|    4|     X|     5|
|     A|    1|     B|     6|
|     B|    2|     B|     6|
|     C|    3|     B|     6|
|     D|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

EJERCICIOS II

- 03.AggregationsSpark
- 04.JoinsSpark



4. Transformaciones Avanzadas

Creación de Ventanas

- Permiten realizar cálculos complejos (rankings, promedios móviles, sumas acumuladas) sobre un conjunto de filas relacionadas.
- A diferencia de groupBy(), mantienen todas las filas originales en el DataFrame; solo añaden una nueva columna con el resultado.
- Definición Clave: Se definen con Window.partitionBy() (la agrupación donde el cálculo se reinicia) y orderBy() (el criterio de ordenación dentro de esa agrupación).

```
windowDF = df.withColumn("ranking",
    row_number().over(Window.partitionBy("Category").orderBy(col("Sales").desc())))
```

Aquí, se crea un ranking de ventas, pero el ranking se reinicia cada vez que cambia el valor de la columna "Category"

Ejemplo de Ventanas

df_ejemplo

Departamento	Empleado	Ventas
A	Juan	100
A	Ana	150
A	Luis	100
B	Marta	200
B	Carlos	200
B	Elena	50
C	Pedro	1000
C	Sofia	1500
C	Maria	1000
C	Andres	500

df_resultado

Departamento	Empleado	Ventas	Ranking_Ventas_Dept
A	Ana	150	1
A	Juan	100	2
A	Luis	100	2
B	Marta	200	1
B	Carlos	200	1
B	Elena	50	3
C	Sofia	1500	1
C	Pedro	1000	2
C	Maria	1000	2
C	Andres	500	4

```
ventana_ranking= Window.partitionBy("Departamento").orderBy(col("Ventas").desc())
df_resultado = df_ejemplo.withColumn("Ranking_Ventas_Dept",rank().over(ventana_ranking))
df_resultado.orderBy("Departamento", col("Ventas").desc()).show()
```

Creación de UDFs

- En spark también se nos permite generar nuestras propias funciones

df

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

```
twoChar = udf(lambda s: s[:2])
twoDF = df.withColumn("two", twoChar(col("letter")))
twoDF.show()
```

twoDF

```
+------+-----+---+
|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 = df.withColumn("plus1", add_one(col("value")))
plus1DF.show()
```

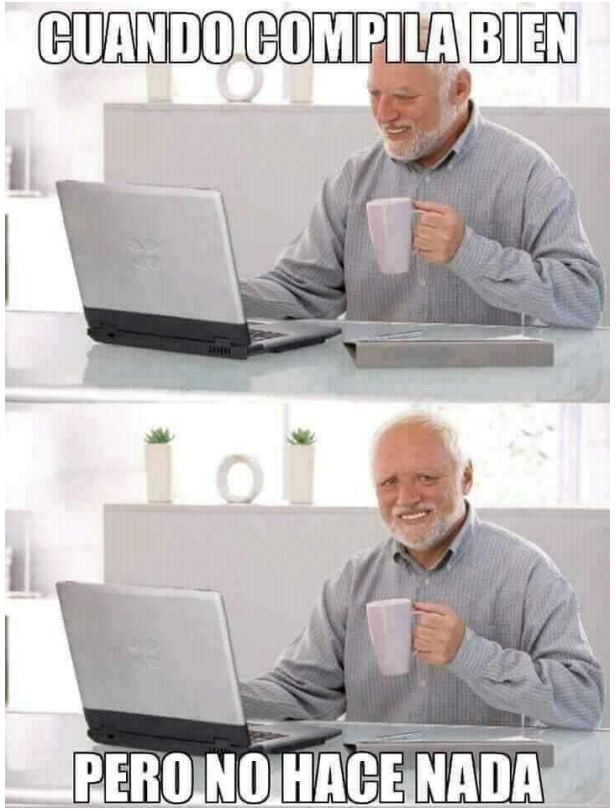
plus1DF

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

- Sin embargo, siempre que sea posible, es preferible utilizar las transformaciones integradas de Spark, especialmente cuando se trabaja con Python.

EJERCICIOS III

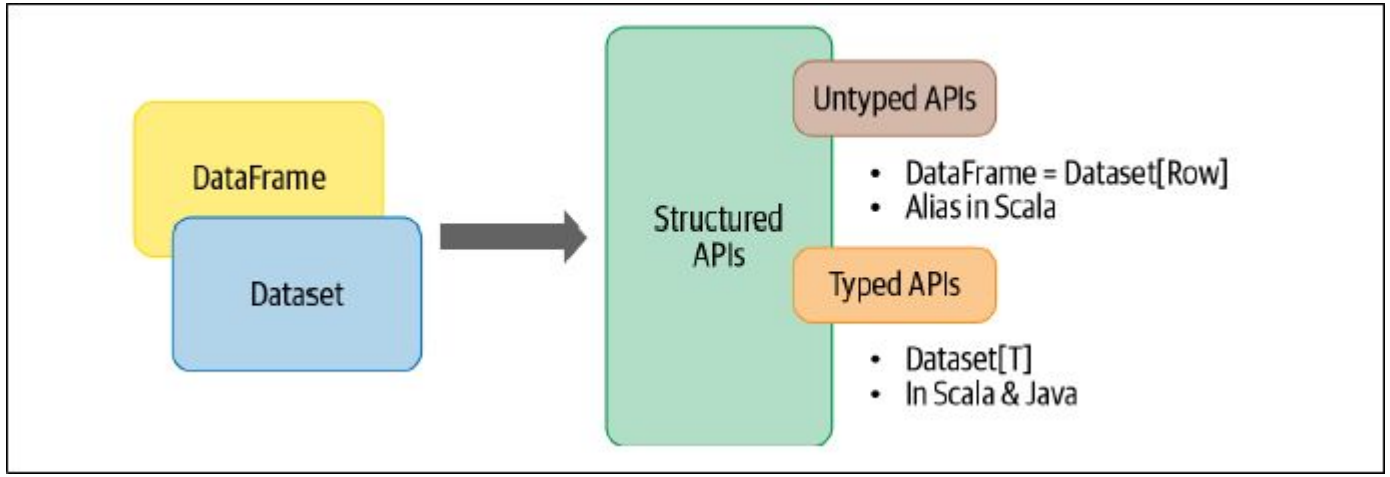
- 05.WindowsAndUDFsSpark



5. DataSet API

DataSet API

- Un dataset es una colección de objetos JVM fuertemente tipados en Scala o una clase en Java y sólo tienen sentido en estos lenguajes de programación.
- En cambio, en Python y R solo se utilizan DataFrames, ya que estos lenguajes son dinámicamente tipados: los tipos de datos se determinan en tiempo de ejecución, no en compilación. Por ello, los Datasets no tienen sentido en estos lenguajes.
- Por último, en Scala, un DataFrame puede considerarse simplemente como un Dataset de filas, es decir, un Dataset[Row], donde cada fila es un objeto genérico sin un tipo definido más allá de su estructura interna.



Creando DataSets

- Al crear un dataset en Scala, la forma más sencilla de especificar el esquema de este, es utilizar una case class, por su parte en Java, se utilizan clases JavaBean.
- En este ejemplo, recibimos un .json de un dispositivo IoT con los siguientes datos y generamos la case class correspondiente especificando su esquema.
- Por último podemos de esta forma generar el dataset infiriendo el esquema que hemos generado previamente

```
{"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}
```

```
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)
```

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

Transformaciones y Acciones con DataSets

- Cabe destacar que, mientras que con DataFrames se expresan las condiciones filter() como operaciones DSL similares a SQL qué son independientes del lenguaje, con Datasets utilizamos expresiones nativas del lenguaje como código Scala o Java.

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


PySpark - Ignacio Reyes Vázquez