

# Conceptos y Aplicaciones de Big Data

---

SPARK SQL

Prof. Waldo Hasperué  
[whasperue@lidi.info.unlp.edu.ar](mailto:whasperue@lidi.info.unlp.edu.ar)

# Temario

---

## Spark SQL

- DataFrames
- UDFs

# ¿Qué imprime?

---

```
caconp = cajasDeAhorro.join(prestamos)
clconp = caconp.map(lambda t: (t[1][0][0],
                               t[1][1][0] * t[1][1][1]))
ddporcl = clconp.reduceByKey(lambda t1, t2: t1 + t2)
dds = ddporcl.join(clientes)
dds = dds.map(lambda t: (t[1][0], (t[1][1][0],
                                   t[1][1][1], t[1][1][2])))
dds = dds.sortByKey(False)
print(dds.take(10))
```

# Spark SQL

---

Spark SQL es una interface que permite usar Spark con datos estructurados, es decir con datos que tienen un esquema.

Permite el uso del lenguaje SQL para realizar tareas de selección, filtrado, agrupación y joins.

Es posible cargar datos de diferentes formatos: JSON, Hive, Parquet, ODBC, JDBC, etc.

Permite una fácil integración entre SQL y Python/Scala/Java

Spark SQL puede ser integrado con HiveQL.

# Spark SQL

---

```
from pyspark import SparkContext
from pyspark.sql import SQLContext, Row

sc = SparkContext("local", "My program")
sqlContext = SQLContext(sc)
```



Con el SparkContext se crea un SQLContext

# Spark SQL - DataFrames

---

Spark SQL trabaja con una abstracción de las RDD, los DataFrames.

Un DataFrame es una RDD con esquema, es decir tuplas de datos con nombres y tipo de datos de cada campo.

- Estos DataFrames son distribuidos

Se crean mediante el *sqlContext*.

# Creando DataFrames desde RDDs

---

```
cliente = sc.textFile("Clientes")
```

```
cliente = cliente.map(lambda t :  
                        t.split("\t"))
```

```
cliente = cliente.map(lambda t :  
                        int(t[0]), t[1] + " " + t[2],  
                        int(t[3]), t[4], t[5])
```

# Creando DataFrames desde RDDs

---

```
cliente = sc.textFile("Clientes")
```

```
cliente = cliente.map(lambda t :  
                        t.split("\t"))
```

```
cliente = cliente.map(lambda t :  
    Row(id = int(t[0]),  
        nom_y_ape = t[1] + " " + t[2],  
        dni = int(t[3]), fecha = t[4],  
        nacionalidad = t[5]) )
```



# Creando DataFrames desde RDDs

---

```
cliente = sc.textFile("Clientes")
```

```
cliente
```

Por cada tupla de la RDD *cliente* estamos creando un objeto Row, donde los campos ahora tienen un nombre.

```
cliente = cliente.map(lambda t :
```


```
    Row(id = int(t[0]),  
        nom_y_ape = t[1] + " " + t[2],  
        dni = int(t[3]), fecha = t[4],  
        nacionalidad = t[5]) )
```

Cliente sigue  
siendo una RDD

# Creando DataFrames desde RDDs

---

```
clienteDF = sqlContext.createDataFrame(cliente)
```



A partir de una RDD con filas "*Row*" le pedimos al sqlContext que nos cree un DataFrame

# DataFrame

---

SQLContext permite la carga de datos estructurados devolviendo los datos en un DataFrame.

```
clienteDF = sqlContext.jsonFile("Clientes_json")
```

```
cajaDeAhorroDF = sqlContext.jsonFile("CajaAhorro_json")
```

```
prestamoDF = sqlContext.jsonFile("Prestamo_json")
```

jsonFile devuelve un DataFrame que es un tipo especial de RDD.

Permite también la carga de datos desde XML, Parquet, bases de datos, etc.

Los DataFrames de Spark son distribuidos porque son una abstracción de una RDD

# DataFrame - API

---

Los DataFrames de Spark tienen su propia API

- ❖ agg
- ❖ cache
- ❖ collect
- ❖ count
- ❖ crossJoin
- ❖ distinct
- ❖ drop
- ❖ filter
- ❖ intersect
- ❖ join
- ❖ orderBy
- ❖ repartition
- ❖ sample
- ❖ take
- ❖ transform
- ❖ union

# Visualizando un DataFrame

---

❖ printschema

root

```
-- apellido: string (nullable = true)
-- dni: long (nullable = true)
-- id: string (nullable = true)
-- nombre: string (nullable = true)
```

❖ show

```
+-----+-----+-----+-----+
|apellido|    dni|    id| nombre|
+-----+-----+-----+-----+
|  Diusz |38210138| 1196|  Xpbyq|
|  Jesggp|35077530|22754| Imusjev|
|  Sjhppbo|11942096|98089|  Fdsthi|
|Twwsyfzt|34296816|31083| Bgqtigp|
|Nwcefuhu|38407701|62977|  Ulzvu|
|  Jpiszv|26155040|56166|Pmdnkvrq|
|  Ceggef|17588621|10709|Gyumtqtr|
|  Kulfyt|37866446|61892|  Rdede|
|  Oecid |25747695|47377| Tazkxqe|
|  Qmwim |14353975|68210|  Xbcqu|
|  Zyzhyqx|24262628|23032|Lkogeftk|
|  Ikmyrnt|39328724|81559|Tjdwjxyb|
|  Vfkzwxm|20173327|51308|  Sgczyu|
```

# Spark SQL

---

```
clienteDF.registerTempTable("Cliente")
cajaDeAhorroDF.registerTempTable("CajaDeAhorro")
prestamoDF.registerTempTable("Prestamo")
```

```
result1 = sqlContext.sql(
    "SELECT cantidad_cuotas
    FROM Prestamo
    WHERE cuota > 20000")

result2 = sqlContext.sql(
    "SELECT nombre, apellido
    FROM Cliente
    WHERE nacionalidad = 'ARG' ")
```

Para hacer uso del nombre de una tabla hay que registrar dicho nombre a partir de un DataFrame creado anteriormente

# Spark SQL

El resultado de una consulta SQL es devuelto como un DataFrame

```
argentinos = sqlContext.sql(  
    "SELECT nombre, apellido  
    FROM Cliente  
    WHERE nacionalidad = 'ARG' ")
```

```
argentinos.registerTempTable("Argentinos")
```

Cualquier DataFrame puede ser registrado como tabla, ya sea porque se levantó de una fuente de datos o porque es el resultado de una consulta

# Spark SQL

---

Un DataFrame es una abstracción de una RDD, por lo tanto se le pueden aplicar todas las funciones de los RDD:

- first, count, take, collect,
- map, filter, reduce
- reduceByKey, countByKey
- etc.

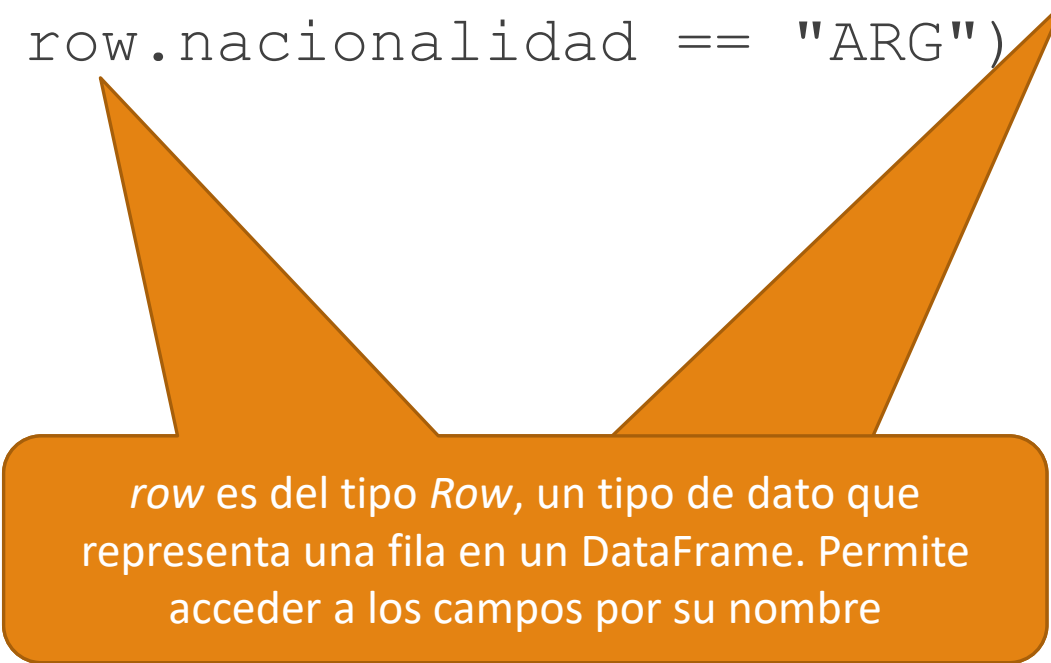


# SparkSQL

---

Por lo tanto es posible hacer el siguiente filtro:

```
res = clientesDF.rdd.filter(lambda row:  
                             row.nacionalidad == "ARG")
```



*row* es del tipo *Row*, un tipo de dato que representa una fila en un *DataFrame*. Permite acceder a los campos por su nombre

# SparkSQL - Ejemplo

---

```
deudas = sqlContext.sql(  
    "SELECT nom_y_ape, dni,  
        sum(P.moneto) AS prestado  
    FROM Prestamo AS P INNER JOIN  
        CajaDeAhorro AS CA ON  
        P.id_caja = CA.id  
        INNER JOIN Cliente AS Cl ON  
        CA.id_cliente = Cl.id  
    GROUP BY nom_y_ape, dni  
    SORT BY prestado DESC  
    LIMIT 10")  
  
deudas.registerTempTable("deudas")
```

# Otra forma de hacer lo mismo...

---

```
caconp = cajasDeAhorroDF.join(prestamosDF,  
                             cajasDeAhorroDF.id == prestamosDF.id_caja)  
clconp = caconp.rdd.map(lambda t: (t.id_cliente,  
                                   t.monto))  
ddporcl = clconp.reduceByKey(lambda t1, t2: t1+t2)  
ddporcl = ddporcl.map(lambda t: Row(id = t[0],  
                                     prestado = t[1]))  
ddporcl = sqlc.createDataFrame(ddporcl)  
dds = ddporcl.join(clientesDF,  
                   ddporcl.id == clientesDF.id)  
dds = dds.sort(dds.prestado.desc())  
print(dds.take(10))
```

# Otra forma de hacer lo mismo...

---

```
caconp = cajasDeAhorroDF.join(prestamosDF,  
                              cajasDeAhorroDF.id == prestamosDF.id_caja)  
clconp = caconp.select(caconp.id_cliente,  
                      caconp.monto)  
ddporcl = clconp.groupBy(clconp.id)  
                      .sum('prestado')  
dds = ddporcl.join(clientesDF,  
                  ddporcl.id == clientesDF.id)  
dds = dds.withColumnRenamed('sum(prestado)',  
                             'prestado')  
dds = dds.sort(dds.prestado.desc())  
print(dds.take(10))
```

# Window

---

Spark SQL tiene lo que llama funciones window, las cuales permiten hacer resúmenes agrupados; calcular la distribución acumulativa, la media móvil o tener acceso a las tuplas anteriores o siguientes de una tupla.

- Aggregate: min, max, avg, count, and sum.
- Ranking: rank, dense\_rank, percent\_rank, row\_num, and ntile
- Analítica: cume\_dist, lag, and lead
- Límites: rangeBetween and rowsBetween

# Window

---

```
from pyspark.sql import Window
```

```
windowPorPais = Window.partitionBy("nacionalidad")
```

Devuelve un objeto *WindowSpec* que dice como agrupar las filas para la tarea de agregación

Podemos indicar más de un campo para hacer la agrupación

# Window - Aggregate

---

```
mn_mx_pais = clientesDF
    .withColumn("más joven",
                F.min("edad")
                    .over(windowPorPais))
    .withColumn("más viejo",
                F.max("edad")
                    .over(windowPorPais))
```

# Window - Aggregate

id	nom_y_ape	dni	edad	nac	más joven	más viejo
12338	Uewme Phclbzlz	34030369	44	BRA	17	80
22940	Cbuglryy Dmbvtsh	31736900	25	BRA	17	80
695	Mwulsqj Psngg	37734451	61	BRA	17	80
91939	Tnmtx Zlkdh	31580690	66	BRA	17	80
75272	Pzctm Xnhra	13377588	70	BRA	17	80
23726	Vkijruji Yvacwzz	18501247	37	BRA	17	80
24042	Obdxwcir Axprljax	13223624	41	BRA	17	80
65097	Dpyaeo Qarwn	15276382	36	BRA	17	80
2580	Qodwq Qwoj	38238668	29	BRA	17	80
37580	Gbhwx b Dmlnshu	25001559	36	BRA	17	80
28086	Mkyqd Bslxk	22417489	43	BRA	17	80
85593	Yoohis Rlfrnjvd	25240617	57	BRA	17	80
98846	Ibycoyi Qcaqniqt	12460339	37	BRA	17	80
20660	Woanu Dekdye	35564644	25	BRA	17	80



# Window - Aggregate

```
from pyspark.sql import functions as F
```

```
mn_mx_pais = clientesDF.withColumn("más joven",  
                                   F.min("edad")  
                                   .over(windowPorPais))  
                        .withColumn("más viejo",  
                                   F.max("edad")  
                                   .over(windowPorPais))  
                        .select("nac", "más joven",  
                               "más viejo")  
                        .dropDuplicates()
```

F.min  
F.max  
F.count  
F.avg  
F.sum

```
min_max_porPais.show()
```

nac	más joven	más viejo
BRA	17	80
BOL	18	72
ITA	23	80
PAR	17	80

# Window - Ranking

---

```
windowPorIDCliente = Window.partitionBy("id_cliente")  
                        .orderBy(F.desc("saldo"))
```

```
rank_clientes = cajasDeAhorroDF.withColumn("Ranking",  
                                           F.rank().over(windowPorIDCliente))
```

```
rank_clientes.show()
```

id	id_cliente	saldo	Ranking
514580	10436	76191.7580739557	1
751567	10436	50541.8251969581	2
778401	10436	21482.614950036	3
757410	10436	-46712.73312844	4
51471	11078	93409.6399663992	1
137016	11078	17025.1015187358	2
994597	11078	-22039.3661977907	3
549776	11078	-95839.6230804918	4

# Window - Ranking

```
windowPorIDCliente = Window.partitionBy("id_cliente")  
                        .orderBy(F.desc("saldo"))
```

```
rank_clientes = cajasDeAhorroDF.withColumn("Ranking",  
                                           F.rank().over(windowPorIDCliente))
```

```
rank_clientes.show()
```

ID	saldo	Rank	Dense	nº row	ntile	saldo
1	9	1	1	1	1	0.0
1	5	2	2	2	1	0.3333333333333333
1	3	3	3	3	2	0.6666666666666666
1	1	4	4	4	2	1.0
2	5	1	1	1	1	0.0
2	4	2	2	2	1	0.25
2	4	2	2	3	1	0.25
2	2	4	3	4	2	0.75
2	1	5	4	5	2	1.0

F.dense\_rank  
F.row\_number  
F.percent\_rank  
F.ntile(n)

# Window - Analítica

```
windowPorIDCliente = Window.partitionBy("id_cliente")  
                        .orderBy(F.desc("saldo"))
```

```
rank_clientes = cajasDeAhorroDF.withColumn("Anterior",  
                                           F.lag("id", 1).over(windowPorIDCliente))  
rank_clientes.show()
```

F.lead

id	id_cliente	saldo	Anterior
514580	10436	76191.7580739557	null
751567	10436	50541.8251969581	514580
778401	10436	21482.614950036	751567
757410	10436	-46712.73312844	778401
51471	11078	93409.6399663992	null
137016	11078	17025.1015187358	51471

# Window personalizada

```
windowPorPaisYEdad = Window.partitionBy("nac")  
                        .orderBy(F.desc("edad"))  
                        .rangeBetween(0, 3)  
  
count_porPais = clientesDF.withColumn("cuantos 1",  
                                       F.count("edad").over(windowPorPaisYEdad))  
  
count_porPais.show()
```

id	nom_y_ape	dni	edad	nac	cuantos 1
73296	Zsazzqo Yenxro	27307629	78	BRA	3
41558	Ukggbd Fquooo	25402567	77	BRA	2
38895	Csmxhrrb Ijnhw	14031735	75	BRA	1
99467	Tqykm Urdqwe	13154086	71	BRA	2
10081	Fkegffi Ayiht	17859726	69	BRA	2
34280	Qxvti Yztcn	26696886	67	BRA	1
39164	Nftjpmv Juidovt	11408857	61	BRA	2
82678	Tvhhvh Fieno	20210688	50	BRA	1

# UDFs (User Defined Functions)

---

¿Qué hace este script?

```
def validar(dni):  
    if (chequeos_pertinentes):  
        return True  
    else:  
        return False  
  
dni_invalidos = clientes.filter(lambda t:  
                                not validar(t['dni']))
```

# UDFs (User Defined Functions)

---

Se puede usar la función *validar* dentro de una cláusula SQL.

```
dni_invalidos = sqlContext.sql("
    SELECT *
    FROM Cliente
    WHERE NOT validarEnSQL(dni)")
```

Se debe registrar la función, vía SQLContext, indicando el valor de retorno.

```
sqlContext.udf.register("validarEnSQL", validar,
                        BooleanType())
```

# UDFs (User Defined Functions)

---

También como operador en operaciones con DataFrames

```
from pyspark.sql.functions import udf
from pyspark.sql.types import BooleanType

invalidos = clientesDF.filter(not_validar(clientes.dni))
check = clientesDF.withColumn("DNI_valido", validar(clientes.dni))

def _validar(dni):
    return (len(str(dni)) == 8)

validar = udf(_validar, BooleanType())

def _not_validar(dni):
    return not _validar(dni)

not_validar = udf(_not_validar, BooleanType())
```