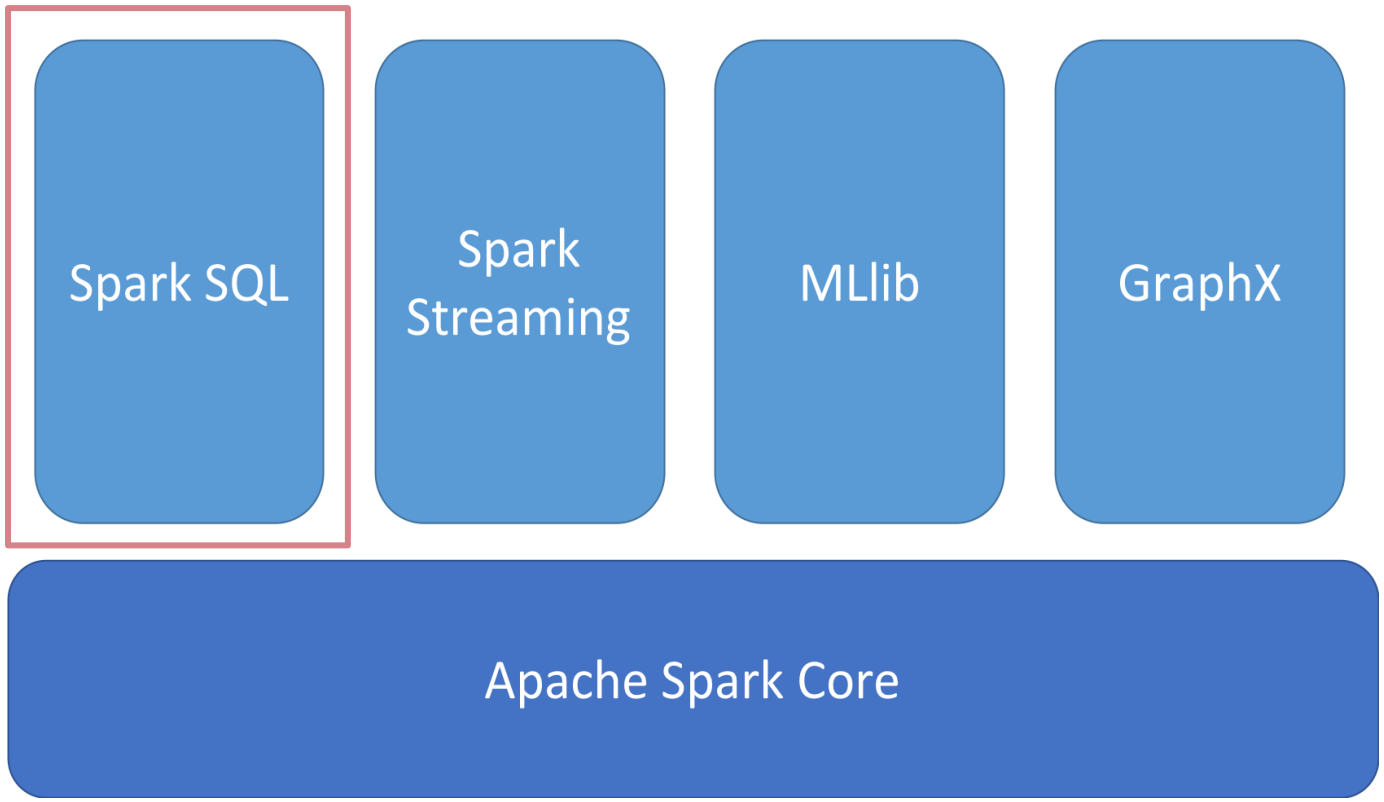
API DATOS ESTRUCTURADOS SPARK

APIS DISPONIBLES EN SPARK

APIs generales disponibles

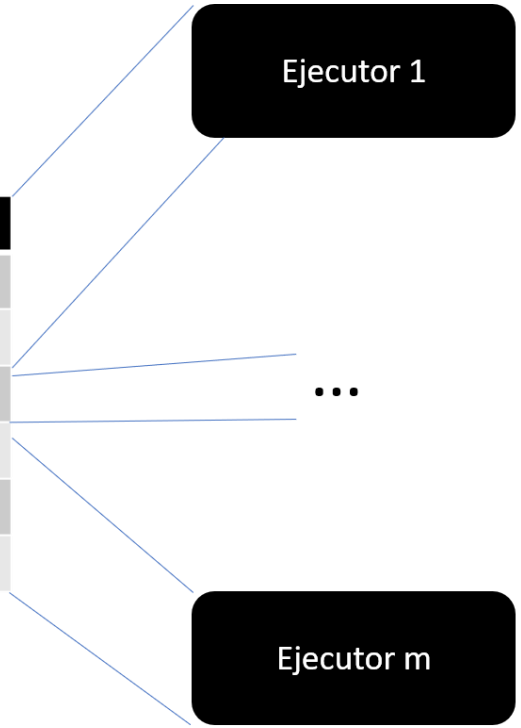


- API Core da **flexibilidad**, pero es **minimalista** → Datos no estructurados
- API Core **no es muy cómodo** para trabajar con datos estructurados.
- Spark cuenta con una API para trabajar con datos estructurados → Dataframes + Datasets + Spark SQL
- **Datasets** → Solo disponibles en Scala y Java
- **Dataframes** y SQL → También en Python y R

DATAFRAME

ABSTRACCIÓN DE DATOS

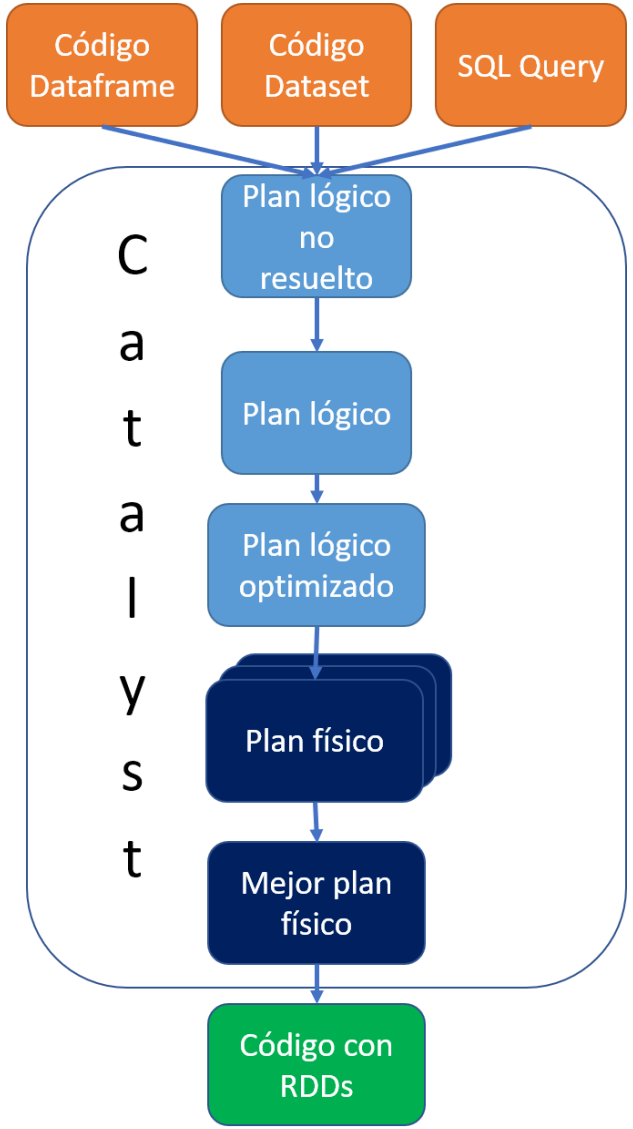
id	nombre	apellido	género	edad
1	John	Doe	M	28
2	Mary	Doe	F	null
…	…	…	…	…
n-2	Peter	Parker	M	33
n-1	Lara	Croft	F	28
n	Linus	Torvalds	M	50



- Es una abstracción de datos que representa una **tabla distribuida**
- ¡OJO! No confundir con los dataframes de Pandas
- La tabla tiene **columnas y filas, además de un esquema** (tipo de datos asociado a las columnas y nombres de columna)
- La distribución se realiza **a nivel de filas**, estando éstas distribuidas entre los ejecutores del clúster
- ¿Qué ofrece con respecto a otros dataframes?
  - Distribución y **cómputo paralelo**
  - **Integración** resto de APIs de Spark
- Está disponible en **Scala, Java, Python y R**

MOTOR CATALYST

OPTIMIZADOR API DATOS ESTRUCTURADOS



- Catalyst se encarga de **optimizar** el trabajo realizado por Spark cuando ejecutamos **código con datos estructurados** (dataframe, dataset, SQL)
- Pasos:
  1. Cuando se requiere un resultado, el código se pasa a Catalyst
  2. El código del usuario es transformado a un **plan lógico no resuelto**, donde aparecen las **transformaciones a llevar a cabo** pero no cómo estas deben ser llevadas a cabo en el clúster. Existen **referencias sin resolver.**
  3. Tras ello, se aplica el Catalog para **resolver referencias no resueltas** en el plan. En esta fase se pueden detectar referencias sin resolver y lanzar error. El resultado es un **plan lógico**.
  4. Tras ello, se aplican reglas para **reducir operaciones redundantes** o **minimizar el número de transformaciones**, aunque sin considerar los recursos del clúster. El resultado es el **plan lógico optimizado**.
  5. Se generan diversos **planes físicos**, los cuales **asignan recursos del clúster** a las tareas.
  6. Se aplica un **estimador de costes** a cada plan físico y se **selecciona el más** adecuado.
  7. Las tareas son del plan físico son **transformadas** a transformaciones sobre **RDDs**.

RDDS, DATAFRAMES Y DATASETS

Ventajas y desventajas

Colección	Ventajas	Desventajas
RDDs	Representación flexible y apropiada para datos no estructurados (e.g., texto). Control total sobre las transformaciones. Hay bastante código legado escrito en RDDs	No se optimiza con Catalyst. Operaciones básicas y de bajo nivel Código más difícil de mantener
Dataframes	Optimizado por Catalyst, ganando ampliamente en rendimiento. API para datos estructurados. Código más mantenible. La mayoría de APIs trabajan ahora con DFs y DS	Poco control sobre cómo el clúster trabaja con el DF. No es posible conocer en tiempo de compilación si una operación o transformación es válida.
Datasets	Optimizado por Catalyst, ganando ampliamente en rendimiento. API para datos estructurados. Código más mantenible. La mayoría de APIs trabajan ahora con DFs y DS	Poco control sobre cómo el clúster trabaja con el DF. Algo de rendimiento es sacrificado a cambio de conocer en tiempo de compilación si una operación o transformación es válida.

EXTRACCIÓN

Operaciones de lectura

EXTRACCIÓN DE DATOS: FICHEROS CSV

- Ejemplo de lectura local

```
app_df = spark.read.csv("file:///your_path/googleplaystore.csv",
      header=True, inferSchema=True, nullValue = "", sep=",")
```

- *header* → Indica si la primera fila contiene la cabecera o nombres de columnas
- *inferSchema* → Indica a Spark si debería intentar inferir los tipos de datos de cada columna del Dataframe (DF)
- *nullValue* → Cómo son especificados los valores nulos en el conjunto de datos
- *sep* → Qué carácter se utiliza para separar las columnas en el fichero
- *encoding* → Codificación del fichero. (UTF-8 por defecto)
- *nanValue* → Qué cadena se emplea para indicar NaN
- *samplingRatio* → Únicamente usado con inferSchema. Porcentaje (0 a 1) del conjunto de datos empleado para inferir el esquema

¿Y SI CONOCEMOS EL ESQUEMA?

- Creación del esquema

```
from pyspark.sql.types import *
schema = StructType( [
      StructField('App', StringType() ),
      StructField('Category', StringType()),
      StructField('Rating', DoubleType()),
      StructField('Reviews', IntegerType()),


])
```

- Lectura con esquema ya creado

```
app_df = spark.read.csv("file:///your_path/googleplaystore.csv",
      header=True, schema=schema, nullValue = "", sep=",")
```

TIPOS DE DATOS EN LOS ESQUEMAS

Tipo de datos esquema	Equivalente Python
ByteType	int o long en el rango -128 a 128
ShortType	int o long en el rango -32768 a 32767
IntegerType	int o long
LongType	int o long en el rango -9223372036854775808 a 9233720368547758O7
FloatType	float
DoubleType	float
DecimalType	decimal.Decimal
BinaryType	bytearray
BooleanType	bool
TimestampType	datetime.datetime
DateType	datetime.date
ArrayType	list, tuple, o array
MapType	dict
StructType	list o tuple

EXTRACCIÓN DE DATOS: FICHEROS JSON

- Ejemplo de lectura local

```
json_df = spark.read.json("file:///your_path/sample_json.json",
multiLine=False)
```

- *schema* → Objeto de tipo Schema, en caso de que lo queramos proporcionar
- *multiLine* → Indica si tenemos un único objeto JSON (True) o si hay un objeto JSON por file (False)
- *encoding* → Codificación del fichero. (UTF-8 por defecto)
- *samplingRatio* → Porcentaje (0 a 1) del conjunto de datos empleado para inferir el esquema

LECTURA DE OTRAS FUENTES DE DATOS

- **Parquet:** Formato orientado a columnas, cuyo almacenamiento hacen muy eficiente el procesamiento por columnas y su almacenamiento. Puede tanto representar tablas planas como datos anidados. Apache Spark recomienda trabajar con este tipo de formato por estar altamente optimizado en Spark.
- **ORC:** Otro formato orientado a columna, normalmente enfocado a trabajar con Hive.
- **JDBC/ODBC:** Conexión directa con una base de datos relacional para lectura y escritura de datos.
- **Texto plano**

TRANSFORMACIÓN

Algunas operaciones importantes

ALGUNOS VIEJOS CONOCIDOS

- *take*
- *collect*
- *distinct*
- *count*
- *sample*
- *union*
- *intersect*
- *subtract*
- *randomSplit*

SELECCIÓN DE COLUMNAS Y CREACIÓN DE NUEVAS

- Ejemplo de selección de varias columnas

```
from pyspark.sql.functions import col, column, expr
new_df1 = app_df.select('Category', 'Rating')
new_df2 = app_df.select( expr('*'), (column('Rating')>3).alias('New Rating') )
new_df3 = app_df.select( col('Category'), col('Rating') )
new_df4 = app_df.select( expr('*'), expr('Rating>3 AS New_Rating') )
new_df5 = app_df.selectExpr( '*', 'Rating>3 AS New_Rating' )
```

- *expr* se emplea para construer una expresión combinando columnas
- *col* y *column* son métodos que usamos para hacer referencia a un nombre de columna en el DF con el que estamos trabajando
- *select* se emplea para seleccionar columnas de un DF, así como para generar nuevas columnas derivadas de las existentes.
- *selectExpr* funciona como select, pero los argumentos son evaluados con *expr* directamente

ALGUNAS EXPRESIONES

- app_df.Price – 30
- col('Price') – 30
- column('Price') – 30
- expr('Price – 30')
- col('Rating')*col('Reviews')
- column('Rating')*column('Reviews')
- expr('Rating*Reviews')
- (col('Price')>0) & (col('Rating')>3)
- (column('Price')>0) & (column('Rating')>3)
- expr('Price>0 AND Rating>3')
- (col('Price')>0) | (col('Rating')>3)
- (column('Price')>0) | (column('Rating')>3)
- expr('Price>0 OR Rating>3')
- expr('NOT Rating > 4')
- ~(column('Rating')>4)
- expr('Rating = 4.1')
- col('Rating')==4.1

SELECCIÓN DE COLUMNAS Y CREACIÓN DE NUEVAS

- Creación de nueva columna, manteniendo el resto intactas

```
new_df = app_df.withColumn("NewCol", expr('Rating>4.0') )
```

Toma como argumentos el nombre de la nueva columna y a qué debe evaluarse la columna

- Sobreescribir el valor de una columna, manteniendo el resto intactas

```
app_df = app_df.withColumn("Price", expr("double(replace(Price,'\$',''))") )
```

- Funciones disponibles dentro de expr en SPARK SQL: *https://spark.apache.org/docs/latest/api/sql/*

RENOMBRAMIENTO DE COLUMNAS

- Renombrado de columnas

```
new_df = app_df.withColumnRenamed("Installs", "Downloads" )
```

El primer parámetro es el nombre que tiene actualmente la columna, y el segundo parámetro es el nombre que le queremos dar a la columna. El resto de columnas se mantienen intactas

ELIMINAR COLUMNAS

- Diciendo explícitamente con qué columnas quedarse

```
new_df = app_df.select("App", "Rating" )
```

- Diciendo explícitamente cuáles queremos eliminar

```
new_df = app_df.drop("Content Rating", "Last Updated" )
```

FILTRADO DE FILAS

- A veces también es interesante filtrar a nivel de filas, quedándose únicamente con aquellas que cumplen una determinada condición.

```
new_df = app_df.filter( expr('Rating>4 AND Price>0') )
```

El parámetro de este método es una expresión que se evalúa a cierto o falso y que opera a nivel de columna

FILTRADO DE FILAS II

- También se pueden filtrar las filas de un DF atendiendo a un número de filas máximo.

```
limit_df = app_df.limit( 15 )
```

Se queda con las n primeras filas, donde n es el número especificado por parámetro. Se suele emplear junto a las operaciones de ordenación.

FILTRADO DE FILAS III

- A veces nos puede interesar eliminar las filas duplicadas. La comparación se hace a nivel de todas las columnas

```
distinct_df = app_df.distinct()
```

- Determinando qué columnas se usan para determiner si las filas son duplicadas

```
distinct_df = reviews_df.dropDuplicates(subset=['App', 'Sentiment'])
```

LIDIANDO CON LOS VALORES NULOS

- A veces nos puede interesar eliminar las filas que tienen valores nulos. Hay diversas formas de alcanzar este cometido:
  - Las filas con algún valor de tipo nulo (en cualquier columna)

```
df1= app_df.dropna('any')
```

  - Las filas con TODOS los valores de tipo nulo

```
df2= app_df.dropna('all')
```

  - Las filas con menos de un número determinado de valores NO nulos

```
df3= app_df.dropna(thres=5)
```

  - Restringiendo los valores nulos a un subconjunto de columnas

```
df4= app_df.dropna('any', subset=['Rating', 'Android Ver'])
```

LIDIANDO CON LOS VALORES NULOS II

- En algunas ocasiones querremos rellenar los valores nulos con un determinado valor:

```
df1= app_df.fillna({'Rating':2.5, 'Category':'Unknown'})
```

  - Se pasa por parámetro un diccionario cuyas claves son las columnas a sustituir valores nulos y cuyos valores son los valores que se introducirán en dichas columnas.
  - Si el tipo de datos de la columna y el tipo de valor especificado para imputar no coinciden, se ignora la asignación.

ORDENANDO TABLAS

- A veces nos puede interesar eliminar las filas duplicadas. La comparación se hace a nivel de todas las columnas

```
from pyspark.sql.functions import asc, desc

sort_df1 = app_df.sort(['Rating', 'Reviews'], ascending=[False, False]).limit(5)

sort_df2 = app_df.sort([col('Rating'), col('Reviews')], ascending=[False,
False]).limit(5)

sort_df3 = app_df.sort( [desc('Rating'), desc('Reviews')]  ).limit(5)

sort_df4 = app_df.sort( [col('Rating').desc(), col('Reviews').desc()]  ).limit(5)
```

  - El primer argumento es la columna o conjunto de columnas en base a cuyos valores se ordenará
  - El parámetro opcional ascending indica si el orden debe ser descendente o ascendente
  - Las funciones asc y desc pueden utilizarse para sustituir a ascending
  - Se suele usar en conjunción con limit

FUNCIONES AGREGACIÓN I

- Sirven para agregar la información de una columna, reduciendo a un único valor. Estas funciones se encuentran disponibles tanto en SQL como funciones a importar

Función	Qué calcula
count(columna)	'*' → Todas las filas de la tabla columna → Valores no nulos de dicha columna
countDistinct(columna)	Número de valores distintos de la columna
min(columna), max(columna)	Mínimo/Máximo de la columna
sum(columna)	Suma de los valores no nulos de la columna
sumDistinct(columna)	Suma de todos los valores no nulos distintos de la columna
avg(columna), mean(columna)	Media de los valores no nulos de la columna
var_pop(columna),var_samp(columna), stddev_pop(columna), stddev_pop(columna)	Varianza y desviación típica poblacional/muestral de los valores no nulos de la columna

FUNCIONES AGREGACIÓN II

- Sirven para agregar la información de una columna, reduciendo a un único valor. Estas funciones se encuentran disponibles tanto en SQL como funciones a importar

Función	Qué calcula
skewness(columna)	Coeficiente de asimetría muestral de los valores no nulos de la columna
kurtosis(columna)	Calcula el coeficiente de kurtosis de los valores no nulos de la columna
corr(columna1, columna2)	Calcula el coeficiente de correlación entre ambas columnas
covar_pop(col1,col2), covar_samp(col1,col2)	Calcula la covarianza poblacional/muestral entre ambas columnas
percentile(columna, percentage)	Calcula el percentil especificado por percentage (0 a 1) para los valores de la columna
...	...

FUNCIONES AGREGACIÓN III

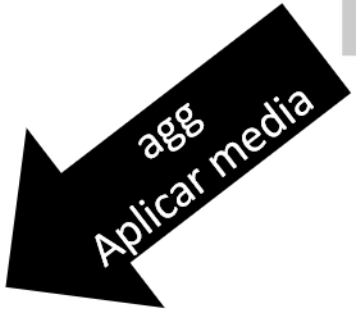
- Ejemplos

```
from pyspark.sql.functions import mean, stddev_samp, kurtosis, skewness

df1 = app_df.select(mean(app_df.Rating), stddev_samp(app_df.Rating),
kurtosis(app_df.Rating), skewness(app_df.Rating), expr('percentile(Rating, 0.5)') )

df1.show()sort_df2 = app_df.sort([col('Rating'), col('Reviews')], ascending=[False,
False]).limit(5)

sort_df3 = app_df.sort( [desc('Rating'), desc('Reviews')]  ).limit(5)

sort_df4 = app_df.sort( [col('Rating').desc(), col('Reviews').desc()]  ).limit(5)
```

AGRUPACIÓN Y AGREGACIÓN

Tratamiento	Temperatura
A	30
B	33
B	28
A	38
B	30

Grupo	Tratamiento	Temperatura
A	A	30
	A	38
B	B	28
	B	33
	B	30



Tratamiento	Temperatura
A	34
B	30.3

AGRUPACIÓN Y AGREGACIÓN

```
grouped_df1 = app_df.groupBy( col('Category')
).agg(expr('mean(Rating)'),expr('min(Rating)'),expr('max(Rating)' ) )


grouped_df2 = app_df.groupBy( col('Category'), col('Content
Rating')).agg(expr('mean(Rating)'),expr('min(Rating)'),expr('max(Rating)' ) )
```

La agrupación se puede hacer a nivel de una columna, o a nivel de varias (Segundo ejemplo).

Los parámetros de agg siempre deben ser una o varias expresiones que devuelvan un único valor

TABLAS DE CONTINGENCIA

Count

		Gender		
		Men	Women	Total
College major	Humanities	4	10	14
	Natural Sciences	11	10	21
	Social Sciences	8	14	22
Total		23	34	57

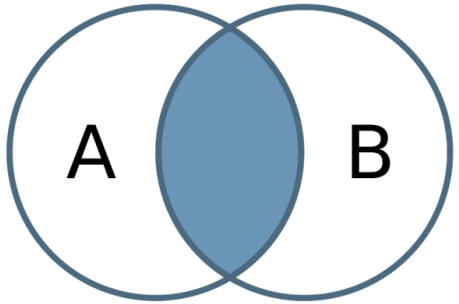
ENLACE DE DATAFRAMES

- Es una operación algo más compleja que en el caso de los RDD. El resultado es un nuevo RDD que contiene las columnas de ambos DFs. Las filas de ambos DFs son unidas por una condición especificada en el momento de la llamada. La condición puede ser simple o compleja.

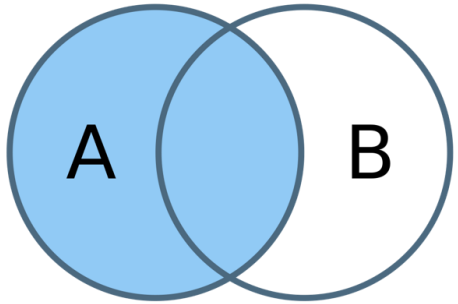
```
joined_df = df1.join(df2, df1.id1 == df2.id2, how='left')


condition = [ df1.id1 == df2.id1, df1.id2 == df2.id2 ]

type_join = 'inner'

joined_df2 = df1.join(df2, on=condition, how=type_join)
```

ENLACE DE DATAFRAMES

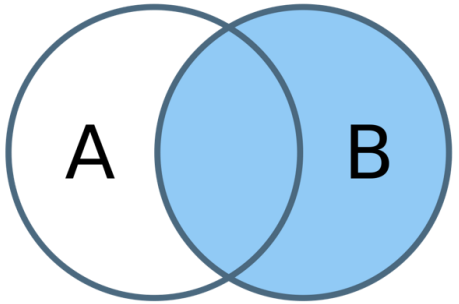
- Tipos de join



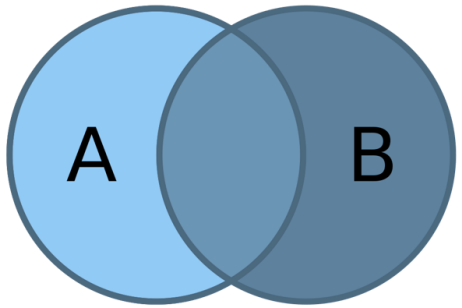
INNER JOIN
Incluye información tanto de A como de B



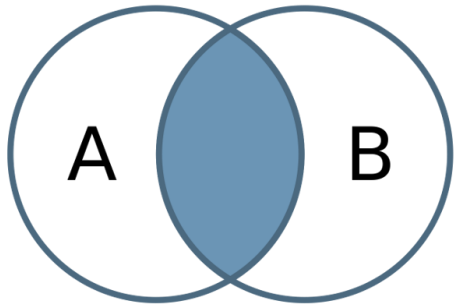
LEFT (OUTER) JOIN
Incluye información tanto de A como de B



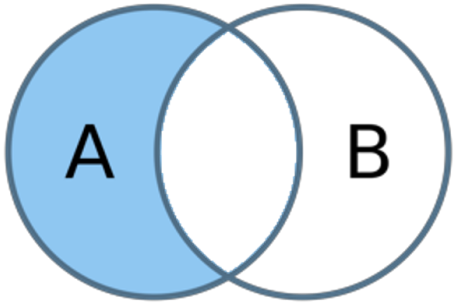
RIGHT (OUTER) JOIN
Incluye información tanto de A como de B



(FULL) OUTER JOIN
Incluye información tanto de A como de B



LEFT SEMI JOIN
Incluye únicamente información de A



LEFT ANTI JOIN
Incluye únicamente información de A

Valor de how	Join realizado
inner	Inner join
cross	Producto cartesiano
outer, full, full_outer	Outer join
left_outer, left	Left outer join
right_outer, right	Right outer join
left_semi	Left semi join
left_anti	Left anti join

CARGA

Algunas instrucciones importantes

CARGA DE DATOS

- Carga en un fichero csv

```
app_df.write.csv("file///your_path/", sep="\t", header=True, nullValue="", emptyValue="")
```

- *sep* → Carácter separador
- *header* → Cierto si queremos cabecera para el fichero, falso en caso contrario
- *nullValue* → Valor que representa nulo
- *emptyValue* → Valor que representa valor vacío
- *mode* →

mode	Explicación
append	Añade los contenidos de este Dataframe a los contenidos de la ruta especificada
overwrite	Sobreescribe los datos existentes en la ruta especificada en la operación de escritura
ignore	Ignora la operación si los datos ya existen, aunque no lanza ningún error.
error	Lanza un error si ya existan datos en la ruta indicada en la operación de escritura. Por defecto

CARGA DE DATOS

- Carga en un fichero csv

```
app_df.write.csv("file///your_path/", sep="\t", header=True, nullValue="", emptyValue="")
```

- *sep* → Carácter separador
- *header* → Cierto si queremos cabecera para el fichero, falso en caso contrario
- *nullValue* → Valor que representa nulo
- *emptyValue* → Valor que representa valor vacío
- *mode* →

mode	Explicación
append	Añade los contenidos de este Dataframe a los contenidos de la ruta especificada
overwrite	Sobreescribe los datos existentes en la ruta especificada en la operación de escritura
ignore	Ignora la operación si los datos ya existen, aunque no lanza ningún error.
error	Lanza un error si ya existan datos en la ruta indicada en la operación de escritura. Por defecto

CARGA EN OTROS FORMATOS DE DATOS

- **JSON**
- **Parquet:** Formato orientado a columnas, cuyo almacenamiento hacen muy eficiente el procesamiento por columnas y su almacenamiento. Puede tanto representar tablas planas como datos anidados. Apache Spark recomienda trabajar con este tipo de formato por estar altamente optimizado en Spark.
- **ORC:** Otro formato orientado a columna, normalmente enfocado a trabajar con Hive.
- **JDBC/ODBC:** Conexión directa con una base de datos relacional para lectura y escritura de datos.
- **Texto plano**

COP
UPV

CONTINUARÁ …

Víctor Sánchez Anguix ✉ vicsana1@upv.es

DSIC
DEPARTAMENT DE SISTEMES INFORMÀTICS I COMPUTACIÓ

DEIOAC
VALENCIA

DEPARTAMENTO DE ESTADÍSTICA E INV. OPERAT. APLICADAS Y CALIDAD

UNIVERSITAT POLITÈCNICA DE VALÈNCIA