Operaciones con DataFrames

Veremos distintas operaciones que se pueden hacer con los DataFrames:

- Filtrado de filas
- Ordenación y agrupamiento
- Joins
- Funciones escalares y agregados
- Manejo de tipos complejos
- Funciones de ventana
- Funciones definidas por el usuario

Acabaremos viendo como usar consultas SQL sobre DataFrames

In [ ]:
```
from pyspark import SparkContext
from pyspark.sql import SparkSession
import os

# Elegir el máster de Spark dependiendo de si se ha definido la variable de entorno HADOOP_CONF_DIR o YARN_CONF
SPARK_MASTER: str = 'yarn' if 'HADOOP_CONF_DIR' in os.environ or 'YARN_CONF_DIR' in os.environ else 'local[*]'

# Creamos un objeto SparkSession (o lo obtenemos si ya está creado)
spark: SparkSession = SparkSession \
  .builder \
  .appName("Mi aplicacion") \
  .config("spark.rdd.compress", "true") \
  .config("spark.executor.memory", "3g") \
  .config("spark.driver.memory", "3g") \
  .master(SPARK_MASTER) \
  .getOrCreate()

sc: SparkContext = spark.sparkContext
```

In [ ]:
```
%%sh
rm -rf /tmp/tcdm-public
git clone -b 24-25 --single-branch --depth 1 https://github.com/dsevilla/tcdm-public.git /tmp/tcdm-public
```

In [ ]:
```
# Recupero el DataFrame leyéndolo del formato parquet
from pyspark.sql.dataframe import DataFrame

dfSE: DataFrame = spark.read\
            .format("parquet")\
            .option("mode", "FAILFAST")\
            .load("/tmp/tcdm-public/datos/dfSE.parquet")
dfSE.cache()
```

In [ ]:
```
dfSE.show(5)
dfSE.printSchema()
```

In [ ]:
```
assert(dfSE.count() == 410346)
dfSE.count()
```

Operaciones de filtrado

In [ ]:
```
# Selecciona los post que tengan la palabra Italiano en su cuerpo
from pyspark.sql.column import Column
from pyspark.sql.dataframe import DataFrame
from pyspark.sql.functions import col

colCuerpo: Column = col("Body")
dfConItaliano: DataFrame = dfSE.filter(colCuerpo.like('%Italiano%'))

print("Número de posts con la palabra Italiano: {0}\n"\
      .format(dfConItaliano.count()))

assert(dfConItaliano.count() == 32)

print("Una de las filas")
dfConItaliano.take(1)
```

In [ ]:
```
# Obtenemos las preguntas (PostTypeId == 1) que tienen una respuesta aceptada (AcceptedAnswerId != null)
# Nota: where() es un alias de filter()

postTypeIdCol: Column = col("PostTypeId")
acceptedAnswerIdCol: Column = col("AcceptedAnswerId")

questionsWithAcceptedAnswersDf: DataFrame = (dfSE
```

```
                    .where((postTypeIdCol == 1) & (acceptedAnswerIdCol.isNotNull()))
                    .withColumnRenamed("CreationDate", "Fecha_de_creación"))

print("Número de preguntas con respuesta aceptada: {0}."\
      .format(questionsWithAcceptedAnswersDf.count()))

questionsWithAcceptedAnswersDf.cache()

(questionsWithAcceptedAnswersDf
        .select("Fecha_de_creación", postTypeIdCol.alias("Tipo Post"), acceptedAnswerIdCol)
        .show(truncate=False))
```

In [ ]:
```
# Nos quedamos con las entradas correspondientes a junio de 2016
from datetime import date

fechaCreacionCol: Column = col("Fecha_de_creación")

dfPregConRespAceptJun16: DataFrame = questionsWithAcceptedAnswersDf\
                    .filter((fechaCreacionCol >= date(2016,6,1)) &
                            (fechaCreacionCol <= date(2016,6,30)))

dfPregConRespAceptJun16.select(fechaCreacionCol, postTypeIdCol, acceptedAnswerIdCol)\
        .show(truncate=False)
```

In [ ]:
```
# Añadimos una columna que contenga el ratio entre el número de vistas y el score

colNumVistas: Column = col("ViewCount")
colPuntos: Column = col("Score")
dfPregConRespAceptyRatio: DataFrame = \
    questionsWithAcceptedAnswersDf.withColumn("ratio", colNumVistas/colPuntos)

# Muestra algunas columnas con ratio > 35
colRatio: Column = col("ratio")
(dfPregConRespAceptyRatio.filter(colRatio > 35)
                         .select(fechaCreacionCol, colNumVistas, colPuntos, colRatio)
                         .show(truncate=False))
```

Operaciones de ordenación y agrupamiento

In [ ]:
```
# Ordenamos por viewCount
questionsWithAcceptedAnswersDf.orderBy(colNumVistas.desc())\
                  .select(fechaCreacionCol, colNumVistas)\
                  .show(10, truncate=False)
```

In [ ]:
```
# Creamos una agrupación por la columna OwnerUserId
from pyspark.sql.group import GroupedData

colUserId: Column = col("OwnerUserId")
grupoPorUsuario: GroupedData = questionsWithAcceptedAnswersDf.groupBy(colUserId)
print(type(grupoPorUsuario))
```

In [ ]:
```
print("DataFrame con el número de posts por usuario.")
dfPostPorUsuario: DataFrame = grupoPorUsuario.count()
dfPostPorUsuario.printSchema()

colNPosts: Column = col("count")
dfPostPorUsuario.select(colUserId.alias("Número de usuario"),
                        colNPosts.alias("Número de posts"))\
                .orderBy(colNPosts, ascending=False).show(10)
```

In [ ]:
```
print("DataFrame con la media de vistas por usuario:")
dfAvgPorUsuario: DataFrame = grupoPorUsuario.avg("ViewCount")\
                    .withColumnRenamed("avg(ViewCount)", "Media_vistas")
dfAvgPorUsuario.orderBy("Media_vistas", ascending=False).show(10)
```

In [ ]:
```
# El método agg permite hacer varias operaciones de agrupamiento, expresadas como un diccionario {nombre_column
print("Obtenemos las tablas anteriores con una sola operación.")
dfCountyAvg: DataFrame = grupoPorUsuario.agg({"OwnerUserId":"count", "ViewCount":"avg"})
dfCountyAvg.printSchema()

colCount: Column = col("count(OwnerUserId)")
colMedia: Column = col("avg(ViewCount)")
dfCountyAvg.select(colUserId.alias("Número de usuario"),
                   colCount.alias("Número de posts"),
                   colMedia.alias("Media de vistas"))\
                  .orderBy(colUserId).show()
```

In [ ]:
```
# Agrupación sobre dos columnas
dfSE.groupBy(colUserId, postTypeIdCol)\
    .count()\
    .orderBy(colUserId.asc(), postTypeIdCol.desc())\
    .show()
```

Una descripción de las funciones que se pueden usar con GroupedData está en

https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.GroupedData.html

Extensiones del groupBy

Funciones `rollup` y `cube`

Rollup

Incluye filas adicionales con agregados por la primera columna

In [ ]:
```
# Contar para cada usuario el número de preguntas (PostTypeId = 1) y el número de respuestas (PostTypeId = 2)
rollupPorUsuarioyTipoPost: GroupedData = dfSE.rollup("OwnerUserId", "PostTypeId")
print(type(rollupPorUsuarioyTipoPost))
```

In [ ]:
```
# DataFrame con el número de post por usuario y tipo pregunta
# Los campos a null son de agregación, por ejemplo:
# null null = todos los posts
# 4    null = todos los posts del usuario con id 4
# 4    1    = todos los post de tipo 1 del usuario 4
dfPostPorUsuarioyTipo: DataFrame = rollupPorUsuarioyTipoPost.count()
dfPostPorUsuarioyTipo.printSchema()
dfPostPorUsuarioyTipo.select(colUserId.alias("Número de usuario"),
                             postTypeIdCol.alias("Tipo de post"),
                             colNPosts.alias("Número de posts"))\
                     .orderBy(colUserId, postTypeIdCol)\
                     .show(100)
```

Cubes

Similar al Rollups, pero recorriendo todas las dimensiones

In [ ]:
```
grupoPorUsuarioyTipoPost: GroupedData = dfSE.cube("OwnerUserId", "PostTypeId")
```

In [ ]:
```
# DataFrame con el número de post por usuario y tipo pregunta
# Los campos a null son de agregación, por ejemplo:
# null null = todas los posts
# null 1    = todos los post de tipo 1
# 4    null = todos los posts del usuario con id 4
# 4    1    = todos los post de tipo 1 del usuario 4
dfPostPorUsuarioyTipo: DataFrame = grupoPorUsuarioyTipoPost.count()
dfPostPorUsuarioyTipo.printSchema()
dfPostPorUsuarioyTipo.select(colUserId.alias("Número de usuario"),
                             postTypeIdCol.alias("Tipo de post"),
                             colNPosts.alias("Número de posts"))\
                     .orderBy(colUserId,postTypeIdCol)\
                     .show(100)
```

Joins

Spark ofrece la posibilidad de realizar múltiples tipos de joins

- inner, outer, left outer, right outer, left semi, left anti, cross

In [ ]:
```
# Buscamos unir las preguntas con respuesta aceptada con la respuesta que se ha elegido como aceptada
# Unimos el campo AcceptedAnswerId de las preguntas con el campo id de las respuestas
dfPreguntas: DataFrame = questionsWithAcceptedAnswersDf\
                .select(colUserId, colCuerpo, acceptedAnswerIdCol)\
                .withColumnRenamed("OwnerUserId", "Usuario pregunta")\
                .withColumnRenamed("Body", "Pregunta")\
                .withColumnRenamed("AcceptedAnswerId", "ID Resp Aceptada")

colId: Column = col("Id")
dfRespuestas: DataFrame = dfSE\
                .select(colId, colUserId, colCuerpo)\
                .where(postTypeIdCol == 2)\
                .withColumnRenamed("Id", "ID Respuesta")\
                .withColumnRenamed("OwnerUserId", "Usuario respuesta")\
                .withColumnRenamed("Body", "Respuesta")

nPreguntas: int = dfPreguntas.count()
AnswerCount: int = dfRespuestas.count()
print("Número de preguntas con respuesta aceptada = {0}.".format(nPreguntas))
print("Número de respuestas = {0}.".format(AnswerCount))
```

In [ ]:
```
# Expresión para el join
joinExpression: Column = dfPreguntas["ID Resp Aceptada"] == dfRespuestas["ID Respuesta"]
```

In [ ]:
```
# Inner join
# Solo se incluyen las filas para las que la joinExpression es true
joinType = "inner"
```

```
dfInner: DataFrame = dfPreguntas.join(dfRespuestas, joinExpression, joinType)
nFilas = dfInner.count()
print("Número de filas = {0}.".format(nFilas))
dfInner.show(100)
```

In [ ]:
```
# Outer join
# Incluye todas las filas de ambos DataFrames.
# En el caso de que no haya equivalente el alguno de los DataFrame, se meten nulls
joinType = "outer"
dfOuter: DataFrame = dfPreguntas.join(dfRespuestas, joinExpression, joinType)
nFilas: int = dfOuter.count()
print("Número de filas = {0}.".format(nFilas))
dfOuter.show(100)
```

In [ ]:
```
# Left Outer join
# Incluye todas las filas del DataFrame de la izquierda (primer DataFrame)
# Si no hay equivalencia en el de la derecha, se pone null.
joinType = "left_outer"
dfLOuter: DataFrame = dfPreguntas.join(dfRespuestas, joinExpression, joinType)
nFilas = dfLOuter.count()
print("Número de filas = {0}.".format(nFilas))
dfLOuter.show(100)
```

In [ ]:
```
# Right Outer join
# Incluye todas las filas del DataFrame de la derecha (segundo DataFrame)
# Si no hay equivalencia en el de la izquierda, se pone null.
joinType = "right_outer"
dfROuter: DataFrame = dfPreguntas.join(dfRespuestas, joinExpression, joinType)
nFilas = dfROuter.count()
print("Número de filas = {0}.".format(nFilas))
dfROuter.show(100)
```

In [ ]:
```
# Left Semi join
# El resultado incluyen los valores del primer DataFrame que existen en el segundo
joinType = "left_semi"
dfLSemi: DataFrame = dfRespuestas.join(dfPreguntas, joinExpression, joinType)
nFilas = dfLSemi.count()
print("Número de filas = {0}.".format(nFilas))
dfLSemi.show(100)
```

In [ ]:
```
# Left Anti join
# El resultado incluyen los valores del primer DataFrame que NO existen en el segundo
joinType = "left_anti"
dfLAnti: DataFrame = dfRespuestas.join(dfPreguntas, joinExpression, joinType)
nFilas = dfLAnti.count()
print("Número de filas = {0}.".format(nFilas))
dfLAnti.show(100)
```

In [ ]:
```
# Cross join
# Producto cartesiano, une cada fila del primer DataFrame con todas las del segundo
# NO DEBE USARSE, EXTREMADAMENTE COSTOSO
dfCross: DataFrame = dfRespuestas.crossJoin(dfPreguntas)
# nFilas = dfCross.count()
# print("Número de filas = {0}.".format(nFilas))
# dfCross.show(100)
```

Funciones escalares y agregados

Spark ofrece un ámplio abanico de funciones para operar con los DataFrames:

- Funciones matemáticas: `abs`, `log`, `hypot`, etc.
- Operaciones con strings: `lenght`, `concat`, etc.
- Operaciones con fechas: `year`, `date_add`, etc.
- Operaciones de agregación: `min`, `max`, `count`, `avg`, `sum`, `sumDistinct`, `stddev`, `variance`, `kurtosis`, `skewness`, `first`, `last`, `window`, etc.

Una descripción de estas funciones se puede encontrar en https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/functions.html.

In [ ]:
```
from pyspark.sql.functions import datediff, col
from pyspark.sql import Row

colUltimaActividad: Column = col("LastActivityDate")
fechaCreacionCol: Column = col("Fecha_de_creación")
# Buscamos la pregunta con respuesta aceptada que estuvo más tiempo activa
# (con la mayor diferencia entre los valores de LastActivityDate y Fecha de creacion)
masActiva: Row | None = questionsWithAcceptedAnswersDf\
            .withColumn("tiempoActiva",
                        datediff(colUltimaActividad,
                                 fechaCreacionCol))\
            .orderBy("tiempoActiva", ascending=False)\
```

```
            .head()
print("La pregunta \n\n{0}\n\nestuvo activa {1} días.".\
      format(masActiva.Body.replace("&lt;", "<").replace("&gt;", ">"), masActiva.tiempoActiva))
```

In [ ]:
```
from pyspark.sql.functions import window

# Obtenemos el número de post por semana de cada usuario
# Agrupamos por OwnerUserId y una ventana de fechas de creación de 1 semana
questionsWithAcceptedAnswersDf.groupBy(
                    colUserId, window(fechaCreacionCol, "1 week").alias("Semana"))\
                  .count()\
                  .sort("count", ascending=False)\
                  .show(20,False)
```

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

# Buscar la media y máximo de la columna "Score" de todas las filas y el número total del DataFrame completo.
dfSE.select(F.avg(colPuntos), F.max(colPuntos), F.count(colPuntos)).show()
```

In [ ]:
```
# Otra forma usando describe
dfSE.select(colPuntos).describe().show()
```

Tipos complejos

Spark permite trabajar con tres tipos de datos complejos: `structs`, `arrays` y `maps`.

Structs

DataFrames dentro de DataFrames

In [ ]:
```
from pyspark.sql.functions import struct,col
# Creamos un nuevo DF con una columna que combina dos columnas existentes
colId: Column = col("id")
colNumVistas: Column = col("ViewCount")
colNRespuestas: Column = col("AnswerCount")
dfStruct: DataFrame = dfSE.select(colId, colNumVistas, colNRespuestas,
                struct(colNumVistas, colNRespuestas).alias("Vistas_Respuestas"))
dfStruct.show(5)
```

In [ ]:
```
dfStruct.printSchema()
```

In [ ]:
```
# Obtenemos un campo de la columna compuesta
dfStruct.select(col("Vistas_Respuestas").getField("ViewCount")).show(5)
```

Arrays

Permiten trabajar con datos como si fuera un array Python

Ejemplo

Obtener el número de *tags* para cada pregunta con respuesta aceptada y eliminar los símbolos `<` y `>`

- Las "tags" de cada pregunta se guardan concatenadas, separadas por < y >, codificados como `<` y `>`

`<english-comparison><translation><phrase-request>`

In [ ]:
```
# Obtenemos un DataFrame sin tags nulas
dfSE.show(10)
dfNoNullTags = dfSE.dropna("any", subset=["Tags"])
dfNoNullTags.select("Tags").show(10, False)
```

In [ ]:
```
# Añado una columna con las etiquetas separadas
from pyspark.sql.functions import split,replace
colTags: Column = col("Tags")
dfTags: DataFrame = dfNoNullTags.withColumn("tag_array", split(colTags, "><"))
dfTags.select(col("tag_array")).show(10, False)
```

In [ ]:
```
dfTags.printSchema()
```

In [ ]:
```
from pyspark.sql.functions import size
# Mostramos el número de etiquetas de cada entrada
colTag_array: Column = col("tag_array")
dfTags.select(colTag_array, size(colTag_array)).show(5, False)
```

In [ ]:
```
# Mostramos la segunda etiqueta de cada entrada
dfTags.selectExpr("tag_array", "tag_array[1]").show(5, False)
```

In [ ]:
```
from pyspark.sql.functions import array_contains
```

```
# Miramos si en las tags aparece la palabra "usage"
dfTags.withColumn("Con_usage", array_contains(colTag_array, "<usage"))\
      .select(colTag_array, col("Con_usage")).show(5, False)
```

In [ ]:
```
from pyspark.sql.functions import explode
# Convertimos cada etiqueta en una fila
dfTagsRows: DataFrame = dfTags.withColumn("Tags2", explode(colTag_array))
dfTagsRows.select(colTags, col("Tags2")).show(10, False)
```

In [ ]:
```
# Elimina los símbolos &lt; y &gt; de las etiquetas (de otra forma diferente a la anterior)
from pyspark.sql.functions import regexp_replace
dfTags: DataFrame = dfTagsRows.withColumn("Tags_separadas",
            regexp_replace("Tags2", "[<>]", "")).drop("Tags2")
dfTags.select(colTags, col("Tags_separadas")).show(10, False)
```

In [ ]:
```
# Número de entradas con la etiqueta word-choice
print("Número de entradas con la etiqueta word-choice = {0}."
      .format(dfTags
        .filter(col("Tags_separadas") == "word-choice")
        .count()))
```

Funciones de ventana

Similares a las de funciones de agregación, permiten operar en grupos de filas devolviendo un único valor para cada fila. Esto permite, entre otras cosas:

- Obtener medias móviles
- Calcular sumas acumuladas
- Acceder a los valores de una fila por encima de la actual

Básicamente, una función de ventana (window function) calcula un valor para cada fila de entrada de una tabla en base a un grupo de filas, denominado *frame*.

Como funciones de ventana se puede usar las funciones de agregación ya comentadas y otras funciones adicionales (`cume_dist` , `dense_rank` , `lag` , `lead` , `ntile` , `percent_rank` , `rank` , `row_number`) especificadas como *Window functions* en https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/window.html.

Ejemplo 1

A partir del DataFrame `dfPregConRespAcept` , mostrar la puntuación (columna "Score") máxima por usuario, y, para cada pregunta, la diferencia de su puntuación con el máximo del usuario.

In [ ]:
```
from pyspark.sql.window import Window, WindowSpec

# Especificamos la ventana que particiona las filas por la columna OwnerUserId
ventana: WindowSpec = Window.partitionBy(colUserId)
```

In [ ]:
```
# Creamos una columna con los máximos valores de Score por usuario
colMaxPuntos: Column = F.max(colPuntos).over(ventana)
```

In [ ]:
```
# Obtenemos un nuevo DataFrame incluyendo la puntuación máxima por usuario
# y la diferencia entre este máximo y la puntuación de cada pregunta
questionsWithAcceptedAnswersDf.select(colUserId, colId.alias("Pregunta"),
                          colPuntos, colMaxPuntos.alias("maxPorUsuario"))\
                  .withColumn("Diferencia", colMaxPuntos-colPuntos)\
                  .orderBy(colUserId, colId)\
                  .show(30)
```

Ejemplo 2

Mostrar para cada usuario y pregunta del DataFrame `dfPregConRespAcept` el número de días que pasaron desde la anterior pregunta del usuario hasta la actual, y desde esta hasta la siguiente.

In [ ]:
```
# Especificamos la ventana que particiona las filas por la columna OwnerUserId y las ordena por fecha de creaci
from pyspark.sql.window import WindowSpec

ventana: WindowSpec = Window.partitionBy(colUserId).orderBy(fechaCreacionCol)
```

In [ ]:
```
# Creamos una columna que referencia a la pregunta anterior por fecha
colAnterior: Column = F.lag(fechaCreacionCol, 1).over(ventana)
# Creamos una columna que referencia a la pregunta posterior por fecha
colPosterior: Column = F.lead(fechaCreacionCol, 1).over(ventana)

# Mostramos para cada usuario y pregunta el id de la pregunta anterior y posterior
questionsWithAcceptedAnswersDf.select(colUserId, colId, fechaCreacionCol.alias("Fecha de creación"),
                          F.datediff(fechaCreacionCol,colAnterior).alias("Días desde"),
                          F.datediff(colPosterior,fechaCreacionCol).alias("Días hasta"))\
                  .orderBy(colUserId, colId)\
```

```
                    .show(30, truncate=False)
```

Funciones definidas por el usuario (UDFs)

Si queremos una función que no está implementada, podemos crear nuestra propia función que opere sobre columnas.

- Las UDFs en Python pueden ser bastante ineficientes, debido a la serialización de datos a Python
- Preferible programarlas en Scala o Java (se pueden usar desde Python)

Ejemplo

Usar UDFs para obtener el número de *tags* para cada pregunta y cambiar los `<` y `>` por < y >

- Las "tags" de cada pregunta se guardan concatenadas, separadas por < y >

`<english-comparison><translation><phrase-request>`

Para contar el número de tags, basta con contar el número de apariciones de `<` en el string.

In [ ]:
```
colTags: Column = col("Tags")
# Obtenemos un DataFrame sin tags nulas
dfNoNullTags: DataFrame = dfSE.dropna("any", subset=["Tags"])
```

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

# Se puede hacer de dos formas, o bien con una función o
# con una anotación @udf

# Definimos una función que devuelva el número de &lt; en un string
@udf(returnType=IntegerType())
def udfCuentaTags(tags):
    return tags.count('&lt;')

# Definimos una función que reemplace &lt y &gt por < y >
def reemplazaTags(tags):
    return tags.replace('&lt;', '<').replace('&gt;', '>')

# Creamos udfs a partir de esta última función
udfReemplazaTags= udf(reemplazaTags)
```

In [ ]:
```
dfNoNullTags.select(udfReemplazaTags(colTags).alias("Etiquetas"),\
                    udfCuentaTags(colTags).alias("nEtiquetas"))\
                   .show(truncate=False)
```

In [ ]:
```
# Llamo a las UDFs Scala usando una expresión (si estuvieran definidas en Scala)
#dfNoNullTags.selectExpr("udfReemplazaTagsSc(Tags) AS Etiquetas",
#                        "udfCuentaTagsSc(Tags) AS nEtiquetas")\
#                  .show(truncate=False)
```

Uso de sentencias SQL

Las sentencias SQL ejecutadas desde Spark se trasladan a operaciones sobre DataFrames

- Se pueden ejecutar sentencias remotas a través del servidor JDBC/ODBC Thrift
- También puede trabajar con datos almacenados en Apache Hive

Para usar sentencias SQL sobre un DataFrame, este tiene que registrarse como una *tabla* o *vista*

- la vista puede crearse como temporal (desaparece al terminar la sesión) o global (se mantiene entre sesiones)

In [ ]:
```
# Registra el DataFrame dfPregConRespAcept como una vista temporal
questionsWithAcceptedAnswersDf\
    .createOrReplaceTempView("tabla_PregConRespAcept")

# Crea una tabla con los datops guardados en Parquet
spark.sql("""CREATE TABLE tabla_SE
             USING PARQUET OPTIONS (path '/tmp/tcdm-public/datos/dfSE.parquet')""")
```

In [ ]:
```
spark.sql("SELECT * FROM tabla_SE").printSchema()
```

In [ ]:
```
# Ejecuta un comando SQL sobre la tabla
dfUser100: DataFrame = spark.sql("""SELECT OwnerUserId,Id FROM tabla_SE
                        WHERE OwnerUserId >= 100""")
dfUser100.show(5)
```

In [ ]:
```
# Podemos ver las tablas creadas
spark.sql("SHOW TABLES").show()
```

In [ ]:
```
# Podemos crear un nuevo DataFrame a partir de una de la tablas
dfFromTable = spark.sql("SELECT * FROM tabla_PregConRespAcept")
dfFromTable.show(5)
```

In [ ]:
```
spark.sql("DROP TABLE IF EXISTS tabla_PregConRespAcept")
spark.sql("DROP TABLE IF EXISTS tabla_SE")

spark.sql("SHOW TABLES").show()
```