Introducción a los DataFrames

En este tema veremos:

- Cómo crear un DataFrame
- Algunas operaciones básicas sobre DataFrames
  - Mostrar filas
  - Seleccionar columnas
  - Renombrar, añadir y eliminar columnas
  - Eliminar valores nulos y filas duplicadas
  - Reemplazar valores
- Guardar los DataFrames en diferentes formatos

Creación de DataFrames

Un DataFrame puede crearse de distintas formas:

- A partir de una secuencia de datos
- A partir de objetos de tipo Row
- A partir de un RDD o DataSet
- Leyendo los datos de un fichero
  - Igual que Hadoop, Spark soporta diferentes filesystems: local, HDFS, Amazon S3
    - En general, soporta cualquier fuente de datos que se pueda leer con Hadoop
  - Spark puede acceder a diferentes tipos de ficheros: texto plano, CSV, JSON, Parquet, ORC, Sequence, etc
    - Soporta ficheros comprimidos
- Accediendo a bases de datos relacionales o NoSQL
  - MySQL, Postgres, etc. mediante JDBC/ODBC
  - Hive, HBase, Cassandra, MongoDB, AWS Redshift, etc.

Creando DataFrames a partir de una secuencia o lista de datos

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 [ ]:
```
from pprint import pp

pp(sc._conf.getAll())
```

In [ ]:
```
from pyspark.sql.dataframe import DataFrame
from pyspark.sql.functions import col

# Creando un DataFrame desde un rango y añadiéndole dos columnas
df: DataFrame = spark.range(1,7,2).toDF("n")
df.show()

# Añadiendo dos columnas al DataFrame
# La expresión para la columna puede incluir operadores.
df.withColumn("n1", col("n")+1).withColumn("n2", 2*col("n1")).show()
```

In [ ]:
```
# DataFrame a partir de una lista de tuplas
l: list[tuple] = [
     ("Pepe", 5.1, "Aprobado"),
     ("Juan", 4.0, "Suspenso"),
     ("Manuel", None, None)]
dfNotas: DataFrame = spark.createDataFrame(l, schema=["nombre", "nota", "cal"])
dfNotas.show()
```

```
dfNotas.printSchema()
```

Creando DataFrames con esquema

A la hora de crear un DataFrame, es conveniente especificar el esquema del mismo:

- El esquema define los nombres y tipos de datos de las columnas
- Se usa un objeto de tipo `StructType` para definir el nombre y tipo de las columnas, y un objeto de tipo `StructField` para definir el nombre y tipo de una columna
- Los tipos de datos que utiliza Spark están definidos en:
  - Para PySpark: https://spark.apache.org/docs/latest/sql-ref-datatypes.html

In [ ]:
```
from pyspark.sql.types import StructField, StructType, FloatType, StringType
from pyspark.sql import Row

# Definimos el esquema del DataFrame
esquemaNotas = StructType(fields=[
    StructField(name="nombre", dataType=StringType(), nullable=False),
    StructField(name="nota", dataType=FloatType(), nullable=True),
    StructField(name="cal", dataType=StringType(), nullable=True)
    ])

# Creamos el DataFrame a partir de una lista de objetos Row
filas: list[Row] = [
    Row("Pepe", 5.1, "Aprobado"),
    Row("Juan", 4.0, "Suspenso"),
    Row("Manuel", None, None)
    ]

dfNotas: DataFrame = spark.createDataFrame(filas, schema=esquemaNotas)
dfNotas.show()
dfNotas.printSchema()
```

Creando DataFrames a partir de un fichero de texto

Cada línea del fichero se guarda como una fila

In [ ]:
```
%%sh
wget -q https://raw.githubusercontent.com/dsevilla/tcdm-public/refs/heads/24-25/datos/quijote.txt.gz
```

In [ ]:
```
# OJO: Se supone que el usuario que se usa es "luser" y que tiene permisos para escribir en el directorio /user
if SPARK_MASTER == 'yarn':
  !hdfs dfs -put quijote.txt.gz /user/luser/
```

In [ ]:
```
dfQuijote: DataFrame = spark.read.text("quijote.txt.gz")
dfQuijote.show(50, truncate=False)
```

Creando DataFrames a partir de un fichero CSV

Como ejemplo vamos a utilizar el fichero de preguntas y respuestas de Stack Overflow en Español, que hemos utilizado en otras ocasiones. Es un fichero CSV, con unos campos que son:

- `Id` : integer: La identificación de la pregunta o respuesta
- `AcceptedAnswerId` : integer: La identificación de la respuesta aceptada (si existe)
- `AnswerCount` : integer: El número de respuestas
- `Body` : string: El cuerpo de la pregunta o respuesta
- `ClosedDate` : timestamp: Fecha de cierre de la pregunta (si está cerrada)
- `CommentCount` : integer: Número de comentarios
- `CommunityOwnedDate` : timestamp: (no se usará)
- `ContentLicense` : string: Licencia de contenido
- `CreationDate` : timestamp: La fecha de creación
- `FavoriteCount` : integer: Número de favoritos
- `LastActivityDate` : timestamp: (no se usará)
- `LastEditDate` : timestamp: (no se usará)
- `LastEditorDisplayName` : string: (no se usará)
- `LastEditorUserId` : integer: (no se usará)
- `OwnerDisplayName` : string: El nombre del propietario (si se borró el usuario)
- `OwnerUserId` : integer: El identificador del propietario
- `ParentId` : integer: El identificador de la pregunta padre (si es una respuesta)
- `PostTypeId` : integer: El tipo de post (1 = pregunta, 2 = respuesta, etc.)
- `Score` : integer: La puntuación de la pregunta o respuesta

- `Tags` : string: El conjunto de etiquetas
- `Title` : string: El título de la pregunta
- `ViewCount` : integer: El número de visitas

Los campos se encuentran separados por el símbolo `","`, y el carácter de escape de comillas es el propio carácter de comillas.

Leemos el fichero infiriendo el esquema

In [ ]:
```
%%sh
wget -q https://github.com/dsevilla/bd2-data/raw/main/es.stackoverflow/es.stackoverflow.csv.7z.001 -O - > es.st
wget -q https://github.com/dsevilla/bd2-data/raw/main/es.stackoverflow/es.stackoverflow.csv.7z.002 -O - >> es.s
```

In [ ]:
```
%%sh
7zr x -aoa es.stackoverflow.csv.7z Posts.csv
rm es.stackoverflow.csv.7z
```

In [ ]:
```
# OJO: Se supone que el usuario que se usa es "luser" y que tiene permisos para escribir en el directorio /user
if SPARK_MASTER == 'yarn':
  !hdfs dfs -put Posts.csv /user/luser/
```

In [ ]:
```
dfSEInfered: DataFrame = spark.read.format("csv")\
                    .option("mode", "FAILFAST")\
                    .option("sep", ",")\
                    .option("escape", "\"")\
                    .option("inferSchema", "true")\
                    .option("lineSep", "\r\n")\
                    .option("header", "true")\
                    .option("nullValue", "")\
                    .load("Posts.csv")
```

Algunas opciones:

1. `mode` : especifica qué hacer cuando se encuentra registros corruptos
   - `PERMISSIVE` : pone todos los campos a null cuando se encuentra un registro corrupto (valor por defecto)
   - `DROPMALFORMED` : elimina las filas con registros corruptos
   - `FAILFAST` : da un error cuando se encuentra un registro corrupto
2. `sep` : separador entre campos (por defecto ",")
3. `inferSchema` : especifica si se deben inferir el tipo de las columnas (por defecto "false")
4. `lineSep` : separador de líneas (por defecto "\n"). Lo hemos cambiado a "\r\n" porque el fichero se ha creado en Windows, aunque de un warning, funciona correctamente
5. `header` : si "true" se toma la primera fila como cabecera (por defecto "false")
6. `nullValue` : carrácter o cadena que representa un NULL en el fichero (por defecto "")
7. `compression` : topo de compresión utilizada (por defecto "none")

Las opciones son similares para otros tipos de ficheros.

In [ ]:
```
# Vemos 5 filas
dfSEInfered.show(5)
```

In [ ]:
```
# Vemos como se ha inferido el esquema
dfSEInfered.schema
```

In [ ]:
```
# Otra forma de verlo
dfSEInfered.printSchema()
```

Leemos especificando el esquema

El esquema inferido tiene ciertos fallos, como considerar algunos campos como strings cuando deberían ser enteros, o tipos Timestamp en vez de Date. Por ello, vamos a especificar el esquema.

In [ ]:
```
from pyspark.sql.types import StructField, StructType, IntegerType, TimestampType, StringType

# Defino el esquema para los elementos de la tabla
# StructType -> Permite definir un esquema para el DF a partir de una lista de StructFields
# StructField -> Definen el nombre y tipo de cada columna, así como si es nullable o no (campo True)
dfSE_Schema = StructType([StructField('Id', IntegerType(), False),
                          StructField('AcceptedAnswerId', IntegerType(), True),
                          StructField('AnswerCount', IntegerType(), True),
                          StructField('Body', StringType(), True),
                          StructField('ClosedDate', TimestampType(), True),
                          StructField('CommentCount', IntegerType(), True),
                          StructField('CommunityOwnedDate', TimestampType(), True),
                          StructField('ContentLicense', StringType(), True),
                          StructField('CreationDate', TimestampType(), True),
                          StructField('FavoriteCount', IntegerType(), True),
```

```
                        StructField('LastActivityDate', TimestampType(), True),
                        StructField('LastEditDate', TimestampType(), True),
                        StructField('LastEditorDisplayName', StringType(), True),
                        StructField('LastEditorUserId', IntegerType(), True),
                        StructField('OwnerDisplayName', StringType(), True),
                        StructField('OwnerUserId', IntegerType(), True),
                        StructField('ParentId', IntegerType(), True),
                        StructField('PostTypeId', IntegerType(), True),
                        StructField('Score', IntegerType(), True),
                        StructField('Tags', StringType(), True),
                        StructField('Title', StringType(), True),
                        StructField('ViewCount', IntegerType(), True)])

# Creo el DataFrame con el esquema definido
dfSE: DataFrame = spark.read.format("csv")\
                    .option("mode", "FAILFAST")\
                    .option("inferSchema", "false")\
                    .option("sep", ",")\
                    .option("header", "true")\
                    .option("nullValue", "")\
                    .option("lineSep", "\r\n")\
                    .option("escape", "\"")\
                    .schema(dfSE_Schema)\
                    .load("Posts.csv")
dfSE.cache()
```

In [ ]:
```
dfSE.sort("Id").show()
```

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

Operaciones básicas con DataFrames

Mostrar filas

In [ ]:
```
# show(n) permite mostrar las primeras n filas (por defecto, n=20)
dfSE.show(5)
```

In [ ]:
```
# Podemos indicar que no trunque los campos largos
dfSE.show(5, truncate=False)
```

In [ ]:
```
from pyspark.sql import Row

# take(n) devuelve las n primeras filas como una lista Python de objetos Row
lista: list[Row] = dfSE.take(5)
pp(lista[1])
# collect() devuelve todo el DataFrame como una lista Python de objetos Row
# Si el DataFrame es muy grande podría colapsar al Driver
#lista2 = dfSE.collect()
#print(lista2[10])
```

In [ ]:
```
# sample(withReplacement, fraction, seed=None) devuelve un nuevo Dataframe con una fracción de las filas
dfSESampled: DataFrame = dfSE.sample(False, 0.1, seed=None)
print("N de filas original = {0}; n de filas muestreadas = {1}".format(dfSE.count(), dfSESampled.count()))
```

In [ ]:
```
# limit(n) limita a n el número de filas obtenidas
dfSE_10filas: DataFrame = dfSE.sample(False, 0.1, seed=None).limit(10)
print("N de filas muestreadas = {0}".format(dfSE_10filas.count()))
dfSE_10filas.show()
```

Ejecutar una operación sobre cada una de las filas

El método `foreach` aplica una función a cada una de las filas

- El DataFrame no se modifica y no se crea ningún otro DataFrame
- El `foreach` se ejecuta en los workers

In [ ]:
```
from pyspark.sql import Row

def printid(f: Row) -> None:
    print(f["Id"])

dfSE_10filas.foreach(printid)
```

Seleccionar columnas

In [ ]:
```
# Crea un nuevo DataFrame seleccionando columnas por nombre
dfIdBody: DataFrame = dfSE.select("Id", "Body")
dfIdBody.show(5)
```

```
print("El objeto dfIdCuerpo es de tipo {0}.".format(type(dfIdBody)))
```

In [ ]:
```
# Otra forma de indicar a las columnas
dfIdBody2: DataFrame = dfSE.select(dfSE.Id, dfSE.Body)
dfIdBody2.show(5)
```

In [ ]:
```
# También es posible indicar objetos de tipo Column
from pyspark.sql.column import Column
from pyspark.sql.functions import col

colId: Column = col("Id")
colCreaDate: Column = col("CreationDate")
print("El objeto colId es de tipo {0}.".format(type(colId)))
print("El objeto colCreaDate es de tipo {0}.".format(type(colCreaDate)))
```

In [ ]:
```
# Y crear un DataFrame a partir de objetos Column, renombrando columnas
dfIdFechaCuerpo: DataFrame = dfSE.select(colId,
                                colCreaDate.alias("Fecha_Creación"),
                                dfSE.Body.alias("Cuerpo"))
dfIdFechaCuerpo.show(5)
```

In [ ]:
```
from pyspark.sql.functions import expr
# El DataFrame anterior usando expresiones
dfIdFechaCuerpoExpr: DataFrame = dfSE.select(
                            expr("Id AS ID"),
                            expr('CreationDate AS `Fecha_Creación`'),
                            expr("Body AS Cuerpo"))
dfIdFechaCuerpoExpr.show(5)
```

In [ ]:
```
# Se pueden usar expresiones más complejas
dfSE.selectExpr("*", # Selecciona todas las columnas
                "(AnswerCount IS NOT NULL) as respuestaValida").show()
```

Renombrar, añadir y eliminar columnas

In [ ]:
```
# Renombramos la columna creationDate
dfSE: DataFrame = dfSE.withColumnRenamed("CreationDate", "Fecha_de_creación")
dfSE.select("Fecha_de_creación",
            dfSE.ViewCount.alias("Número_de_vistas"),
            "Score",
            "PostTypeId")\
            .show(truncate=False)
```

In [ ]:
```
# Añadimos una nueva columna con todos sus valores iguales a 1
from pyspark.sql.functions import lit
# lit convierte un literal en Python al formato interno de Spark
# (en este ejemplo IntegerType)
dfSE: DataFrame = dfSE.withColumn("unos", lit(1))
dfSE.show(5)
```

In [ ]:
```
# Elimina una columna con drop
dfSE: DataFrame = dfSE.drop(col("unos"))
dfSE.columns
```

Eliminar valores nulos y duplicados

In [ ]:
```
# Eliminamos todas las filas que tengan null en alguna de sus columnas
dfNoNulls: DataFrame = dfSE.dropna("any")
print("Numero de filas inicial: {0}; número de filas sin null: {1}."
       .format(dfSE.count(), dfNoNulls.count()))
```

In [ ]:
```
# Elimina las filas que tengan null en todas sus columnas
dfNingunNull: DataFrame = dfSE.dropna("all")
print("Número de filas con todo a null: {0}."
       .format(dfSE.count() - dfNingunNull.count()))
```

In [ ]:
```
# Elimina las filas duplicadas
dfSinDuplicadas: DataFrame = dfSE.dropDuplicates()
print("Número de filas duplicadas: {0}."
       .format(dfSE.count() - dfSinDuplicadas.count()))
```

In [ ]:
```
# Elimina las filas duplicadas en alguna columna
dfSinUserDuplicado: DataFrame = dfSE.dropDuplicates(["OwnerUserId"])
print("Número de usuarios únicos: {0}."
       .format(dfSinUserDuplicado.count()))
```

In [ ]:
```
# Otros ejemplos
dfNoNullViewCountAcceptedAnswerId: DataFrame = dfSE\
        .dropna("any", subset=["ViewCount", "AcceptedAnswerId"])
print("Número de filas con ViewCount y AcceptedAnswerId no nulo: {0}."
```

```
       .format(dfNoNullViewCountAcceptedAnswerId.count()))
dfNoNullViewCountAcceptedAnswerId = dfSE\
        .dropna("all", subset=["ViewCount", "AcceptedAnswerId"])
print("Número de filas con ViewCount o AcceptedAnswerId no nulo: {0}."
       .format(dfNoNullViewCountAcceptedAnswerId.count()))
```

Reemplazar valores

In [ ]:
```
# Reemplazamos los null en los campos ViewCount y AnswerCount
dfSE: DataFrame = dfSE.fillna(0, subset=["ViewCount", "AnswerCount"])
dfSE.show(5)
```

In [ ]:
```
# Reemplaza el valor 1 por 2 en una columna nueva añadida

dfSE_w_unos: DataFrame = dfSE.withColumn("unos", lit(1))

dfSE_w_unos.select("Id", "unos").show(10)
dfSE_w_unos.replace(1, 2, subset=["unos"])\
    .select("Id", "unos")\
    .show(10)
```

Guardando DataFrames

Al igual que con la lectura, Spark puede guardar los DataFrames en múltiples formatos

- CSV, JSON, Parquet, Hadoop...

También puede escribir en bases de datos

In [ ]:
```
# Guardo el DataFrame dfSE en formato JSON
dfSE.write.format("json")\
    .mode("overwrite")\
    .save("dfSE.json")
```

In [ ]:
```
%%sh
ls -lh dfSE.json
head dfSE.json/part-*.json
```

In [ ]:
```
# Guardo el DataFrame usando Parquet
dfSE.write.format("parquet")\
    .mode("overwrite")\
    .option("compression", "gzip")\
    .save("dfSE.parquet")
```

In [ ]:
```
print(dfSE.rdd.getNumPartitions())
```

In [ ]:
```
%%sh
# Parquet usa por defecto formato comprimido snappy
ls -lh dfSE.parquet
```

Se crean tantos ficheros como particiones tenga el DataFrame

In [ ]:
```
dfSE2: DataFrame = dfSE.repartition(2)
# Guardo el DataFrame  usando Parquet, con compresión gzip
dfSE2.write.format("parquet")\
     .mode("overwrite")\
     .option("compression", "gzip")\
     .save("dfSE2.parquet")
```

In [ ]:
```
%%sh
ls -lh dfSE2.parquet
```

Particionado

Permite particionar los ficheros guardados por el valor de una columna

- Se crea un directorio por cada valor diferente en la columna de particionado
  - Todos los datos asociados a ese valor se guardan en ese directorio
- Permite simplificar el acceso a los valores asociados a una clave

In [ ]:
```
# Guardo el DataFrame particionado por el PostTypeId (usando Parquet)
dfSE.write.format("parquet")\
    .mode("overwrite")\
    .partitionBy("PostTypeId")\
    .save("dfSE-particionado.parquet")
```

In [ ]:
```
%%sh
```

```
ls dfSE-particionado.parquet
ls -lh dfSE-particionado.parquet/PostTypeId=2
rm -rf dfSE-particionado.parquet
```