miad4.png

```
import os
# Por precaución, cambiamos el directorio activo de Python a aquel que
contenga este notebook
if "PAD-book" in os.listdir():
    os.chdir(r"PAD-book/Laboratorio-Computacional-de-Analytics/S5 -
Extraer, transformar y cargar datos/S5.TU3/")
```

Extracción, transformación y carga de datos: `pyspark`

Extraer, transformar y cargar (ETL por sus siglas en inglés), describe un proceso en tres etapas:

1. obtener datos de una o más fuentes;
2. transformar los datos, sea haciéndoles limpieza, combinándolos o añadiendo registros;
3. grabar los datos, sea cargándolos a su misma fuente, persistiéndolos en archivos locales o alimentando consumidores que dependen de los resultados (aplicaciones *downstream*).

Hasta ahora hemos tratado fuentes de datos almacenadas localmente en archivos de texto o formatos de herramientas estadísticas como Stata. La realidad es que los datos existen en un sinfín de contextos, casi siempre, en formatos altamente dedicados que exigen procesos especializados de extracción, como SQL (*Structured Query Language*) para bases de datos relacionales. También nos hemos enfocado solo en consumir los datos sin preocuparnos mucho acerca de cómo podemos hacer nuestros resultados disponibles en ocasiones futuras o para otros usuarios.

Las bases de datos, como las hemos trabajado, existen únicamente en la memoria volátil de nuestro computador y no persisten de una sesión de Python a la siguiente. Podemos hacer persistir los cambios si los grabamos en archivos tipo `.txt` o `.csv`, pero esto no es manejable a escala o, por ejemplo, cuando queremos desagregar los datos y distribuirlos en distintas tablas para que sean compatibles con operaciones del álgebra relacional (lo que hacen las funciones `join` y `merge` de `pandas`).

En este tutorial exploraremos el uso de la librería `pyspark` para el manejo de bases de datos relacionales, en el contexto de procesos de ETL para analítica de datos.

Requisitos

Para desarrollar este tutorial necesitarás:

- Importar y exportar archivos de texto en formato `.txt` o `.csv` por medio de un *file handle*.
- Utilizar operaciones sencillas y vectorizadas en `numpy` y `pandas`.
- Crear, consultar y utilizar métodos para explorar y manipular objetos tipo `DataFrame` en `pandas`.

Objetivos

Al final de este tutorial podrás:

1. Distinguir situaciones en las que resulta más beneficioso utilizar herramientas de ETL como `pyspark`. **2.** Reconocer estructuras de datos que permiten operar en paralelo. **3.** Extraer y transformar tablas de bases de datos relacionales con operaciones de algebra relacional. **4.** Crear y cargar tablas en bases de datos relacionales.

1. Entorno de desarrollo en Apache Spark

La fundación de *software* Apache es una comunidad dedicada al desarrollo de herramientas *open source*, entre estas, Spark: un motor de procesamiento de datos altamente eficiente. Spark está diseñado para desplegarse en entornos de cómputo distribuido (*cluster*) y paraleliza sus operaciones de manera implícita, lo que lo hace ideal para el procesamiento de altos volúmenes de datos (*Big Data*). Cualquier aplicación puede llegar a imlementar Spark en su flujo de datos por medio de sus APIs (*Application Program Interface*) para distintos lenguajes de programación, como `pyspark` para Python o `SparkR` para R.

Resulta útil trabajar con herramientas como Spark en contextos donde paralelizar el procesamiento de los datos representaría un ahorro significativo en el tiempo de ejecución.

Así como Spark es una aplicación externa a Python a la que accedemos por medio de su API, debemos:

- importar la API;
- configurar la sesión;
- inicializar la aplicación.

Nos enfocaremos en el módulo `sql` para procesos de ETL con bases de datos relacionales.

```
# Importamos la clase SparkSession
from pyspark.sql import SparkSession

# Invocamos un constructor de sesiones de Spark SQL
spark = SparkSession.builder

# Aplicamos una configuración básica
spark = spark.master("local[*]")

# Nombramos la instancia
spark = spark.appName("Instancia_Tutorial_PySpark")

# Exigimos a Spark que almacene sus datos en el directorio Archivos.
spark = spark.config("spark.sql.warehouse.dir", "./Archivos/")

# Habilitamos la interacción con archivos de bases de datos
spark = spark.enableHiveSupport()

# Inicializamos la instancia
```

```
spark = spark.getOrCreate()

# Aunque no lo recomendamos para tu desarrollo, apagaremos las
advertencias de este notebook.
spark.sparkContext.setLogLevel("OFF")
spark
```

```
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use
setLogLevel(newLevel).

22/09/15 08:09:21 WARN NativeCodeLoader: Unable to load native-hadoop
library for your platform... using builtin-java classes where
applicable
22/09/15 08:09:21 WARN Utils: Service 'SparkUI' could not bind on port
4040. Attempting port 4041.

<pyspark.sql.session.SparkSession at 0x7f184850d220>
```

El método `master` nos permite definir el contexto de paralelización de los procesos. Veamos algunos parámetros que puede recibir:

- `"local[<n>]"` le indica a la aplicación que debe ejecutarse en el mismo computador, con `n` cantidad de procesos para paralelizar las tareas (`n` = `*` es la máxima cantidad de procesos para el poder de procesamiento disponible);
- `"Yarn"` (u otro nombre de un administrador de *clusters*), le indican a la aplicación que debe ejecutarse en un *cluster*, el cual tendríamos que configurar.

Al hacer clic en el enlace del *output* de la celda (Spark UI), podrás acceder a una interfaz web, igual a la de la imágen, con detalles de la aplicación. Esto funciona solo si expones a la red el puerto de la aplicación o si ejecutas el programa en tu propio computador, ya que necesitas acceso al servicio web desplegado por Spark.

Spark_GUI.png

2. Estructuras de datos en `pyspark`

Para facilitar la ejecución de tareas en paralelo, Spark incluye dos estructuras de datos que permiten distribuir sus operaciones sobre distintos objetos en memoria.

2.1. Objeto RDD (*Resilient Distributed Dataset*)

Un RDD es la representación de una colección de objetos que pueden distribuirse en distintos procesos o incluso almacenarse en distintos nodos de nuestro *cluster*. Resultan bastante útiles y eficientes para procesar grandes cantidades de datos, pero resultan poco convenientes si nuestra intención es hacer cambios sobre los datos, ya que son inmutables (una vez creados, no pueden editarse).

Esta representación es poco restrictiva en cuanto a qué consideramos un objeto de la colección. Para nuestras intenciones, podríamos pensar en un RDD como un `DataFrame` de `pandas` cuyas

filas hemos almacenado en distintas variables. Alternativamente, podríamos particionar sobre las columnas, en cuyo caso tendríamos una colección de objetos tipo `Series` o, si una partición toma más de una columna, una colección de objetos tipo `DataFrame`.

Declaración

Podemos declarar un RDD a partir de diferentes estructuras de datos. A continuación, vemos un ejemplo de cómo declarar uno a partir de una lista de listas:

```
data_list = [["Programa_1", 2000, 4500, "LAN"],
             ["Programa_2", 3401, 7000, "LAS"],
             ["Programa_3", 50,   7000, "LAS"],
             ["Programa_4", 7850, 3300, "USW"],
             ["Programa_5", 8000, 3505, "LAN"]]

rdd = spark.sparkContext.parallelize(data_list)
rdd
```

```
ParallelCollectionRDD[0] at readRDDFromFile at PythonRDD.scala:274
```

Nota que al intentar imprimir el RDD, nos encontramos con que la representación en el *output* de la celda no muestra el contenido. Esto se debe a que las operaciones sobre los RDD son *lazy* (perezosas) y no se ejecutarán hasta que sea absolutamente necesario. Este esquema de ejecución diferida permite optimizar de manera anticipada las consultas y operaciones para minimizar el tiempo de ejecución y la ocupación de la memoria.

Operaciones y consulta

Por ahora, basta con saber que hay acciones y transformaciones. Bajo ninguna circunstancia estas operaciones modificarán el RDD original; en su lugar, Spark crea una copia del RDD con los cambios necesarios una vez se hayan realizado las operaciones cargadas de manera perezosa.

Veámos a continuación un ejemplo de cómo operar sobre los elementos de la colección.

```
# Declaramos una funcion que tome los elementos con indice 1 y 2 de un
iterable y los multiplique.
def mult(x):
    return x[1] * x[2]

# La transformacion map toma cada objeto contenido en el RDD y ejecuta
la funcion que recibe por parametro.
nuevo_rdd = rdd.map(mult)
result = nuevo_rdd.collect()
print(result)
```

```
PythonRDD[1] at RDD at PythonRDD.scala:53
```

La variable `nuevo_rdd` contiene un RDD distinto a `rdd` para el cual aún no se ha efectuado la transformación. Hacemos uso del método `collect` para ejecutar las operaciones pendientes y recolectar los resultados en Python.

```
rdd.collect()
```

```
[['Programa_1', 2000, 4500, 'LAN'],
 ['Programa_2', 3401, 7000, 'LAS'],
 ['Programa_3', 50, 7000, 'LAS'],
 ['Programa_4', 7850, 3300, 'USW'],
 ['Programa_5', 8000, 3505, 'LAN']]
```

```
nuevo_rdd.collect()
```

```
[9000000, 23807000, 350000, 25905000, 28040000]
```

Tener que recolectar los objetos de RDD resulta útil en escenarios donde debemos procesar nuestros datos en distintas etapas y no nos interesa el resultado de operaciones intermedias. La ejecución perezosa de las transformaciones nos permite concatenarlas sin ocupar memoria de manera redundante y optimiza automáticamente e inteligentemente las instrucciones para no repetir tareas.

2.2. Objeto `DataFrame`

Similar a los `DataFrame` en `pandas`, los `DataFrame` en `pyspark` son una colección mutable de datos organizados por columnas. A diferencia de `pandas`, que opera de manera secuencial, la implementación de `pyspark` almacena las filas de manera distribuida y opera sobre ellas en paralelo.

Declaración

Podemos declarar un `DataFrame` a partir de diferentes estructuras de datos. A continuación, vemos un ejemplo de cómo declarar uno a partir de una lista de listas:

```
columns = ["Nombre", "Descargas", "Lineas_de_codigo", "Region"]
df = spark.createDataFrame(data=data_list, schema=columns)
df
```

```
DataFrame[Nombre: string, Descargas: bigint, Lineas_de_codigo: bigint,
Region: string]
```

El parámetro `schema` (esquema) hace referencia a los campos de una estructura de datos. En el caso de nuestra lista de listas, el esquema es el nombre de las columnas. Si tuvieramos un diccionario de listas, el esquema serían sus llaves.

En `pyspark` existen métodos dedicados a cargar datos desde una gran variedad de formatos. Consideremos, por lo pronto, un formato con el que estemos familiarizados.

```
df_covid_19 = spark.read.option("header","true").csv("Archivos/BID-
Cornell.csv")
df_covid_19.printSchema()
```

```
root
 |-- id: string (nullable = true)
```

```
 |-- medios_noti_redessociales: string (nullable = true)
 |-- medios_noti_chat: string (nullable = true)
 |-- medios_noti_periodicos: string (nullable = true)
 |-- medios_noti_tv: string (nullable = true)
 |-- medios_noti_radio: string (nullable = true)
 |-- medios_covid_redessociales: string (nullable = true)
 |-- medios_covid_chat: string (nullable = true)
 |-- medios_covid_periodicos: string (nullable = true)
 |-- medios_covid_tv: string (nullable = true)
 |-- medios_covid_radio: string (nullable = true)
```

Operaciones y consulta

Tenemos distintas alternativas para consultar el contenido de nuestros `DataFrame`. Podemos utilizar el método `show` para imprimir de manera estilizada la tabla o los métodos `head` y `tail` para retornar las primeras o últimas filas de la tabla.

```
df.show()
```

```
+----------+---------+----------------+------+
|    Nombre|Descargas|Lineas_de_codigo|Region|
+----------+---------+----------------+------+
|Programa_1|     2000|            4500|   LAN|
|Programa_2|     3401|            7000|   LAS|
|Programa_3|       50|            7000|   LAS|
|Programa_4|     7850|            3300|   USW|
|Programa_5|     8000|            3505|   LAN|
+----------+---------+----------------+------+
```

```
df.head(3)
```

```
[Row(Nombre='Programa_1', Descargas=2000, Lineas_de_codigo=4500,
Region='LAN'),
 Row(Nombre='Programa_2', Descargas=3401, Lineas_de_codigo=7000,
Region='LAS'),
 Row(Nombre='Programa_3', Descargas=50, Lineas_de_codigo=7000,
Region='LAS')]
```

```
df.tail(2)
```

```
[Row(Nombre='Programa_4', Descargas=7850, Lineas_de_codigo=3300,
Region='USW'),
 Row(Nombre='Programa_5', Descargas=8000, Lineas_de_codigo=3505,
Region='LAN')]
```

Cabe notar que por medio de los métodos `head` y `tail`, obtenemos objetos de tipo `Row` que son los que distribuye Spark para paralelizar las operaciones. Alternativamente, el método `limit` nos permite obtener las primeras filas del `DataFrame`.

```
df.limit(3).show()
```

```
+----------+---------+----------------+------+
|    Nombre|Descargas|Lineas_de_codigo|Region|
+----------+---------+----------------+------+
|Programa_1|     2000|            4500|   LAN|
|Programa_2|     3401|            7000|   LAS|
|Programa_3|       50|            7000|   LAS|
+----------+---------+----------------+------+
```

En ocasiones puede ser deseable convertir los `DataFrame` en `pyspark` a sus equivalentes en `pandas`. Transformar un `DataFrame` de `pyspark` a `pandas` es recomendable solo si se cuenta con suficiente memoria. Los altos volúmenes de datos que manejamos en `pyspark` con gran eficiencia en memoria y procesamiento pueden no ser compatibles con otras librerías.

```
pandas_df = df.toPandas()
pandas_df
```

```
       Nombre  Descargas  Lineas_de_codigo Region
0  Programa_1       2000              4500    LAN
1  Programa_2       3401              7000    LAS
2  Programa_3         50              7000    LAS
3  Programa_4       7850              3300    USW
4  Programa_5       8000              3505    LAN
```

De manera similar a los RDD, podemos operar sobre los registros de un `DataFrame` en `pyspark` con funciones nativas o por medio de funciones definidas por el usuario. El método `withColumn` nos permite agregar o reemplazar una columna en un `DataFrame`, poblándola registro a registro. El resultado es un nuevo `DataFrame` que refleja estos cambios.

```
df.withColumn(colName, col)
```

El parámetro `colName` permite definir el nombre que tendrá la columna en el nuevo `DataFrame`. El parámetro `col` recibe una expresión en función de las columnas existentes a partir de la cual `pyspark` calcula el valor para cada registro de la columna `colName`.

Importamos la función `col` para crear la expresión del parámetro `col`.

```
from pyspark.sql.functions import col
nuevo_df = df.withColumn("DescargasXLineas", col("Descargas") *
col("Lineas_de_codigo"))
nuevo_df.show()
```

```
+----------+---------+----------------+------+----------------+
|    Nombre|Descargas|Lineas_de_codigo|Region|DescargasXLineas|
+----------+---------+----------------+------+----------------+
|Programa_1|     2000|            4500|   LAN|         9000000|
|Programa_2|     3401|            7000|   LAS|        23807000|
```

```
|Programa_3|       50|            7000|   LAS|          350000|
|Programa_4|     7850|            3300|   USW|        25905000|
|Programa_5|     8000|            3505|   LAN|        28040000|
+----------+---------+----------------+------+----------------+
```

Veamos algunas operaciones de `pyspark` cuyo comportamiento es similar al de `pandas`.

Métodos para imputar faltantes

Método `dropna`

Este método elimina aquellas filas que contengan entradas nulas.

```
DataFrame.dropna(how, thresh=0, subset=DataFrame.columns)
```

- `how`:
  - `how = 'any'`: elimina la fila si contiene al menos un faltante.
  - `how = 'all'`: elimina la fila si solo contiene faltantes.
- `thresh`: elimina todas las filas con más datos faltantes que el umbral (de tipo `int`) especificado.
- `subset`: permite seleccionar un subconjunto de columnas sobre el cual aplicar el método.

A continuación, declaramos un `DataFrame` para ejemplificar el uso del método `dropna`.

```
data = [[1   , None, None, 2   ],
        [None, None, 1   , None],
        [None, 0   , None, 2   ]]

columns = ["A", "B", "C", "D"]

df_numeros = spark.createDataFrame(data=data, schema=columns)
df_numeros.show()
```

```
+----+----+----+----+
|   A|   B|   C|   D|
+----+----+----+----+
|   1|null|null|   2|
|null|null|   1|null|
|null|   0|null|   2|
+----+----+----+----+
```

Utilizamos el método `dropna` para eliminar aquellas filas que solo contienen datos faltantes en el subconjunto de columnas `A` y `C`.

```
nuevo_df_numeros = df_numeros.dropna(how="all", subset=["A","C"])
nuevo_df_numeros.show()
```

```
+----+----+----+----+
|   A|   B|   C|   D|
+----+----+----+----+
|   1|null|null|   2|
|null|null|   1|null|
+----+----+----+----+
```

Método `fillna`

Este método reemplaza las entradas nulas.

```
DataFrame.fillna(value=None, method=None, axis=None, inplace=False,
limit=0)
```

- `value`: indica el valor o diccionario de valores para imputar en las entradas nulas.
- `subset`: permite seleccionar un subconjunto de columnas sobre el cual aplicar el método.

Vamos a completar los datos del `DataFrame` `df_numeros`, reemplazando cada valor faltante por un valor predeterminado para su respectiva columna.

```
nuevo_df_numeros = df_numeros.fillna(value={"A": 1, "B": 0, "C": 1,
"D": 2})
nuevo_df_numeros.show()
```

```
+---+---+---+---+
|  A|  B|  C|  D|
+---+---+---+---+
|  1|  0|  1|  2|
|  1|  0|  1|  2|
|  1|  0|  1|  2|
+---+---+---+---+
```

Métodos para unir tablas

Método `join`

Agrega a un `DataFrame` las columnas de otro según coincidan las columnas especificadas en los dos.

```
DataFrame.join(other, on=None, how='inner')
```

- `other`: el `DataFrame` a unir.

- `on`: permite usar una o varias columnas de los `DataFrame` para encontrar las coincidencias. Las columnas especificadas deben existir en ambos `DataFrame`.
- `how`: es `"inner"` por defecto. Puedes consultar los posibles valores en la documentación, aunque por lo general todas las operaciones `join` tienen la misma nomenclatura independiente del programa de manejo de datos.

Consideremos los siguientes `DataFrame`.

```
data = [[1, 2],
        [3, 4],
        [1, 3]]

columns = ["A", "B"]

df1 = spark.createDataFrame(data = data, schema=columns)
df1.show()
```

```
+---+---+
|  A|  B|
+---+---+
|  1|  2|
|  3|  4|
|  1|  3|
+---+---+
```

```
data = [[3, 2, 4],
        [3, 4, 1]]

columns = ["B", "C", "D"]

df2 = spark.createDataFrame(data = data, schema=columns)
df2.show()
```

```
+---+---+---+
|  B|  C|  D|
+---+---+---+
|  3|  2|  4|
|  3|  4|  1|
+---+---+---+
```

Debemos agregar a `df1` las columnas de `df2` según coincidencia de la columna `"B"`, preservando todas las filas de `df1` y solo las filas que coinciden de `df2`. Utilicemos el método `join`, especificando que la unión será por coincidencia izquierda externa.

```
df_join = df1.join(df2, on = "B", how = 'left')
df_join.show()
```

```
+---+---+----+----+
|  B|  A|   C|   D|
+---+---+----+----+
|  2|  1|null|null|
|  4|  3|null|null|
|  3|  1|   4|   1|
|  3|  1|   2|   4|
+---+---+----+----+
```

Métodos de consulta tipo SQL

Método `select`

Por medio del método `select` podemos indicar las columnas de un `DataFrame` que queremos consultar. Retorna un nuevo `DataFrame`.

Método `where`

Permite filtrar las filas de un `DataFrame` según una condición especificada.

Métodos `groupBy` y `agg`

El método `groupBy` permite agrupar los registros de un `DataFrame` según los valores únicos de la combinación de una o más columnas. El método `agg` permite calcular conteos, sumas, promedios y otras operaciones para cada uno de los grupos resultantes de la función `groupBy`.

Método `orderBy`

Ordena los registros de un `DataFrame` a partir de los valores de las columnas especificadas.

Veamos un ejemplo de una consulta elaborada a un `DataFrame`.

Trabajarémos con los datos importados anteriormente en la variable `df_covid_19`. La intención es calcular, para cada combinación única de los valores de las variables `"medios_noti_chat"`, `"medios_noti_periodicos"` y `"medios_noti_tv"`, la cantidad de registros correspondientes. Reportaremos solo aquellas combinaciones con más de 3000 ocurrencias, ordenadas de mayor a menor.

```
from pyspark.sql.functions import count


df_consulta = df_covid_19.groupBy("medios_noti_chat",
"medios_noti_periodicos", "medios_noti_tv")\
                         .agg(count("*").alias("Count")) \
                         .where(col("Count") >= 3000) \
                         .orderBy(col("Count").desc())
df_consulta.show()
```

```
+----------------+---------------------+-------------+-----+
|medios_noti_chat|medios_noti_periodicos|medios_noti_tv|Count|
+----------------+---------------------+-------------+-----+
```

```
|         Siempre|               Siempre|       Siempre|12526|
|         A veces|               A veces|       A veces|10317|
|         A veces|               A veces|  Casi siempre| 9099|
|         Siempre|               A veces|       Siempre| 7980|
|         A veces|                 Nunca|       A veces| 7267|
|    Casi siempre|               A veces|       A veces| 7232|
|    Casi siempre|               A veces|  Casi siempre| 6765|
|         Siempre|               A veces|       A veces| 6578|
|    Casi siempre|          Casi siempre|  Casi siempre| 5811|
|         A veces|          Casi siempre|  Casi siempre| 5610|
|         A veces|                 Nunca|  Casi siempre| 5357|
|         A veces|               A veces|       Siempre| 4860|
|         Siempre|                 Nunca|       Siempre| 4768|
|         Siempre|               A veces|  Casi siempre| 4691|
|    Casi siempre|                 Nunca|       A veces| 4592|
|    Casi siempre|               A veces|       Siempre| 4422|
|         Siempre|                 Nunca|       A veces| 4259|
|         Siempre|          Casi siempre|       Siempre| 4212|
|         A veces|          Casi siempre|       A veces| 4055|
|         A veces|                 Nunca|       Siempre| 3913|
+----------------+----------------------+--------------+-----+
only showing top 20 rows
```

Métodos `createOrReplaceTempView` y `sql`

Ejecuta una consulta sobre un `DataFrame` a partir de un `str` que contiene una consulta escrita en SQL.

La especificación del lenguaje SQL reserva algunas palabras a las cuales les atribuye significados particulares. Una consulta puede resultar en ediciones de la base de datos, aunque por lo general solo hacemos consultas de selección.

Algunas de las palabras reservadas para las consultas de selección son:

- `SELECT`: permite especificar las columnas que deben incluirse en el resultado de la consulta. Funciones como `COUNT` incluyen columnas caluladas en el resultado. Pueden seleccionarse todas las columnas existentes con `*` y `AS` permite nombrar columnas con un alias.
- `FROM`: permite especificar de qué tabla deben extraerse las columnas seleccionadas o calculadas. La tabla puede ser directamente una existente en la base de datos o una construida a partir de varias operaciones.
    - `JOIN` y `ON`: permiten unir tablas según las mismas reglas de unión que ya conocemos. Puede especificarse el tipo de unión con declaraciones como `LEFT JOIN`, `INNER JOIN` o `OUTER JOIN`, entre otros. Incluimos `ON` para indicar una expresión que compare las columnas, con lo cual se decide si se realiza la unión en cada registro.

  - `WHERE`: permite filtrar los registros de la tabla construida hasta el momento, según una expresión lógica que aplica para cada registro de las columnas especificadas. Podemos concatenar condiciones con las palabras `OR` y `AND`.

- `GROUP BY`: realiza la misma operación de agrupación por combinaciones de columnas que conocemos de `pandas` y `pyspark`.
- `HAVING`: permite filtrar de manera similar a `WHERE`, luego de haber agrupado y seleccionado las columnas especificadas con `SELECT` y `GROUP BY`.
- `ORDER BY`: permite ordenar la tabla resultante según sus columnas. Podemos agregar las palabras `ASC` o `DESC` para especificar el orden.

Una consulta de selección puede verse de la siguiente forma:

```
SELECT Columna_1, Columna_2, COUNT(*) AS Conteo
FROM Tabla_1
INNER JOIN Tabla_2 ON Columna_2
WHERE Columna_1 = 'Valor_1'
GROUP BY Columna_1, Columna_2
HAVING Columna_2 != 'Valor_2'
ORDER BY Columna_1 DESC;
```

Aunque `pyspark` es bastante versátil al momento de realizar consultas sobre una base de datos, hay ocasiones en las que resulta beneficioso poder realizar consultas directamente con la sintaxis de SQL. El método `createOrReplaceTempView` hace el `DataFrame` visible al administrador de `SQL`.

Presentamos un ejemplo de cómo podemos realizar la misma consulta sobre el `DataFrame` `df_covid_19`, por medio del método `sql`.

```
df_covid_19.createOrReplaceTempView("df_covid_19")

consulta = \
'''\
SELECT medios_noti_chat, medios_noti_periodicos, medios_noti_tv,
COUNT(*) AS Count
FROM df_covid_19
GROUP BY medios_noti_chat, medios_noti_periodicos, medios_noti_tv
HAVING Count >= 3000
ORDER BY Count DESC;
'''

spark.sql(consulta).show()
```

```
+----------------+---------------------+--------------+-----+
|medios_noti_chat|medios_noti_periodicos|medios_noti_tv|Count|
+----------------+---------------------+--------------+-----+
|         Siempre|               Siempre|       Siempre|12526|
|         A veces|               A veces|       A veces|10317|
```

```
|          A veces|                A veces|  Casi siempre| 9099|
|          Siempre|                A veces|       Siempre| 7980|
|          A veces|                  Nunca|       A veces| 7267|
|    Casi siempre|                A veces|       A veces| 7232|
|    Casi siempre|                A veces|  Casi siempre| 6765|
|          Siempre|                A veces|       A veces| 6578|
|    Casi siempre|          Casi siempre|  Casi siempre| 5811|
|          A veces|          Casi siempre|  Casi siempre| 5610|
|          A veces|                  Nunca|  Casi siempre| 5357|
|          A veces|                A veces|       Siempre| 4860|
|          Siempre|                  Nunca|       Siempre| 4768|
|          Siempre|                A veces|  Casi siempre| 4691|
|    Casi siempre|                  Nunca|       A veces| 4592|
|    Casi siempre|                A veces|       Siempre| 4422|
|          Siempre|                  Nunca|       A veces| 4259|
|          Siempre|          Casi siempre|       Siempre| 4212|
|          A veces|          Casi siempre|       A veces| 4055|
|          A veces|                  Nunca|       Siempre| 3913|
+-----------------+----------------------+--------------+-----+
only showing top 20 rows
```

3. Cargar datos en `pyspark`

Para almacenar nuestros datos de manera eficiente y poder acceder a ellos en el futuro, nos apoyamos en los administradores de bases de datos relacionales (tipo SQL). Spark incluye su propio administrador de bases de datos (Apache Hive) que nos permite interactuar con archivos compatibles con extensión `.db`, aunque pueden configurarse otros administradores.

Utilicemos una consulta de definición para crear una base de datos (llamada `demo_db`) que pueda almacenar varias tablas. Siempre que queramos crear una base de datos o crear dentro de ella una tabla, utilizamos el comando `CREATE` de SQL. La práctica más común es crear una única base de datos y almacenar en esta todas las tablas que queramos almacenar.

```
consulta = \
'''\
CREATE DATABASE IF NOT EXISTS demo_db;
'''

_ = spark.sql(consulta)
```

`IF NOT EXISTS` nos permite evitar comportamiento inesperado. En caso de que ya exista un objeto del mismo tipo que queremos crear, con el nombre especificado, la consulta no tendrá efecto.

Podemos consultar las bases de datos disponibles para `pyspark` con la consulta `SHOW DATABASES`. El resultado es un `DataFrame` cuya única columna es `namespace` y cuyos valores son los nombres de las bases de datos. La base de datos `default` es temporal y se borra con el final de la sesión, mientras que las otras persisten en almacenamiento.

```
consulta = \
'''\
SHOW DATABASES;
'''

spark.sql(consulta).show()
```

```
+---------+
|namespace|
+---------+
|  default|
|  demo_db|
+---------+
```

Luego de crear la base de datos, encontraremos un archivo o directorio llamado `demo_db.db` en nuestra carpeta de archivos.

```
import glob


# Retorna una lista con la ruta relativa a cualquier directorio
# o archivo terminado en `".db"`, dentro de la carpeta `"Archivos"`.
glob.glob("Archivos/*.db")
```

```
['Archivos/demo_db.db']
```

Cuando creamos una tabla, debemos especificar qué columnas va a tener y el tipo de datos que van a almacenar. Para que las bases de datos de SQL sean eficientes, necesitan conocimiento absoluto acerca de cuánto espacio van a ocupar sus datos. Introducimos una tabla vacía con nombre `tabla_1`, columnas `id`, `name` y `age`, de tipos `INT` (entero), `STRING` (cadena de texto) e `INT`, respectivamente. Para garantizar que la tabla se crea dentro de la base de datos, seguimos el patrón: `<database_name>.<table_name>`.

```
consulta = \
'''\
CREATE TABLE IF NOT EXISTS demo_db.tabla_1
   (id INT,
    name STRING,
    age INT)
STORED AS ORC;
'''

_ = spark.sql(consulta)
```

Validamos que existe la tabla en la base de datos, aunque esta no contenga registros.

```
consulta = \
'''\
```

```
SELECT *
FROM demo_db.tabla_1;
'''

spark.sql(consulta).show()
```

```
+---+----+---+
| id|name|age|
+---+----+---+
+---+----+---+
```

Utilizamos el comando `INSERT INTO` para ingresar nuevas filas en una tabla.

```
consulta = \
'''\
INSERT INTO demo_db.tabla_1
VALUES (1, 'Camilo Gomez', NULL);
'''

_ = spark.sql(consulta)
```

Volvemos a consultar la tabla para validar que agregamos el registro correctamente.

```
consulta = \
'''\
SELECT *
FROM demo_db.tabla_1;
'''

spark.sql(consulta).show()
```

```
+---+------------+----+
| id|        name| age|
+---+------------+----+
|  1|Camilo Gomez|null|
+---+------------+----+
```

En lugar de crear una tabla registro por registro, podemos utilizar los `DataFrame` con los que venimos trabajando que ya existen en `pyspark`. Por ejemplo, podemos ingresar a nuestra base de datos la tabla `df_covid_19` (recuerda que esta la hicimos visible a SQL por medio del método `createOrReplaceTempView`).

```
consulta = \
'''\
CREATE TABLE IF NOT EXISTS demo_db.df_covid_19 STORED AS ORC
    AS SELECT * FROM df_covid_19;
'''
```

```
_ = spark.sql(consulta)
```

Validamos que la tabla existe en la base de datos.

```
consulta = \
'''\
SELECT *
FROM demo_db.df_covid_19;
'''

spark.sql(consulta).limit(3).show()
```

```
+---------+-------------------------+----------------
+----------------------+-------------+-----------------
+--------------------------+-----------------+----------------------
+---------------+------------------+
|       id|medios_noti_redessociales|medios_noti_chat|
medios_noti_periodicos|medios_noti_tv|medios_noti_radio|
medios_covid_redessociales|medios_covid_chat|medios_covid_periodicos|
medios_covid_tv|medios_covid_radio|
+---------+-------------------------+----------------
+----------------------+-------------+-----------------
+--------------------------+-----------------+----------------------
+---------------+------------------+
|4022452.0|                  Siempre|         Siempre|
Nunca|       Siempre|          Siempre|                     Siempre|
Casi siempre|                   Nunca|        Siempre|
Siempre|
|4013528.0|                  Siempre|         Siempre|
Nunca|       Siempre|          Siempre|                     Siempre|
Siempre|                   Nunca|        Siempre|           Siempre|
|4005685.0|                  Siempre|         Siempre|
Siempre|       Siempre|             Siempre|                      Siempre|
Siempre|                 Siempre|        Siempre|           Siempre|
+---------+-------------------------+----------------
+----------------------+-------------+-----------------
+--------------------------+-----------------+----------------------
+---------------+------------------+
```

Ahora conocemos nuevas formas en las que podemos interactuar con nuestros datos en Python. En breve, podemos extraer datos de casi cualquier formato (incluyendo bases de datos de SQL), transformar sus representaciones en Python como `DataFrame` y cargarlos para ocasiones futuras en otra o la misma base de datos de SQL.

Referencias

SparkBy{Examples}. Spark with Python (PySpark) Tutorial For Beginners. Recuperado el 12 de Agosto de 2022 de: https://sparkbyexamples.com/pyspark-tutorial/

Apache PySpark. pyspark.sql.DataFrame. Recuperado el 12 de Agosto de 2022 de: https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.DataFrame.html

Créditos

Autores: Alejandro Mantilla Redondo, Diego Alejandro Cely Gómez

Fecha última actualización: 09/09/2024 Jose Fernando Barrera