2 - Tema 2 (Scala)

December 23, 2020

1 2. Introducción:

1.1 2.1. Spark Session

La aplicación de Spark se controla a través de un controlador llamado SparkSession. La instancia SparkSession es la forma en que se ejecuta Spark y el código definido por el usuario. Siempre debemos iniciar una instancia SparkSession al principio:

[1]:
```
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.functions._
```

```
Intitializing Scala interpreter …

Spark Web UI available at http://DESKTOP-9VTH92L:4041
SparkContext available as 'sc' (version = 3.0.1, master = local[*], app id =␣
↪local-1608742832268)
SparkSession available as 'spark'
```

[1]:
```
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.functions._
```

Si queremos saber la versión que estamos utilizando de Spark podemos hacer:

[2]:
```
print(spark.sparkContext.version)
```

```
3.0.1
```

Crearemos una simple tarea que consistirá en la creación de un rango de números que será como una columna de números en una hoja de cálculo:

[3]:
```
val mi_Rango = spark.range(1000).toDF("numero")
```

[3]:
```
mi_Rango: org.apache.spark.sql.DataFrame = [numero: bigint]
```

1.2 2.2. DataFrames:

Un DataFrame es la API estructurada más común y simplemente representa una tabla de datos con filas y columnas. La lista que define las columnas y los tipos dentro de esas columnas se llama el esquema. Se puede pensar en un DataFrame como una hoja de cálculo de columnas con nombre. La diferencia fundamental es que una hoja de cálculo se encuentra en una computadora, mientras que un Spark DataFrame puede estar dividida en miles de computadoras. La razón para poner los datos en más de un ordenador es sencillo, los datos son demasiado grandes para caber en una sola máquina o simplemente llevaría demasiado tiempo realizar ese cálculo en una sola máquina.

2.2.1. Particiones: Para que cada executor pueda realizar su trabajo en paralelo con el resto, Spark divide los datos en trozos más pequeños llamados particiones. Una partición es una colección de filas que están en una máquina física en el clúster. Las particiones del DataFrame representan cómo se distribuyen físicamente los datos en las máquinas del cluster durante la ejecución. Si solo se tiene una partición, Spark tendrá un paralelismo de solo uno, aunque tengamos miles de executors. De la misma forma si tenemos muchas particiones pero, solo un executor, Spark seguirá teniendo un paralelismo de solo uno.

Una cosa importante a tener en cuenta es que cuando trabajamos con DataFrames no manipulamos las particiones de forma manual (la mayoría de las veces). Simplemente definimos las transformaciones que queremos y Spark determina como se ejecutará a nivel interno en el clúster. Existe API de nivel inferior a través de RDDs que veremos más adelante.

1.3 2.3. Transformaciones:

Spark utiliza lo que se conoce como evaluación perezosa. Esto significa que no hará ningún trabajo, a menos que realmente tenga que hacerlo. Este enfoque permite evitar el uso innecesario de recursos, además de permitir una mejor optimización del proceso. Las llamadas transformaciones, se evalúan de forma perezosa y solo se ejecutará realmente cuando se produzca una acción que definiremos en breve.

[4]:
```
val divisoresDe2 = mi_Rango.where("numero % 2 = 0")
```

[4]:
```
divisoresDe2: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [numero:
bigint]
```

Como podemos observar en el código anterior, no hemos obtenido ningún resultado. Esto es porque solo hemos especificado una transformación y Spark no la ejecutará hasta que ejecutemos una acción. Digamos que Spark va "apuntando" las diferentes transformaciones que deseamos hacer sobre el DataFrame, una vez ejecutemos una acción optimizará todas las diferentes transformaciones y las llevara a cabo.

Hay que diferenciar dos tipos de transformaciones, las que se dice que tienen dependencias estrechas, **Narrow transformations**, y las que tienen amplias dependencias, **Wide transformations (Shuffles)**.

Las transformaciones Narrow como se puede ver en la imagen, son aquellas en las que cada partición de entrada contribuirá a una sola partición de salida. En el anterior fragmento de código, se trata

de un tipo Narrow, donde la "filtración" de números divisores de 2 pueden hacerse en cada partición (o nodo) de forma independiente sin intercambio de datos.

Por otra parte, una transformación Shuffle, tendrá particiones de entrada que contribuyen a varias particiones de salida. Podemos pensar por ejemplo en transformaciones que necesiten hacer operaciones entre columnas y por lo tanto tenga que haber intercambio de datos entre las diferentes particiones (o nodos).

En las operaciones de tipo Narrow, Spark las ejecuta por medio de pipelines de manera que si especificamos varios filtros, los realizará todos en memoria. En el caso de las transformaciones de tipo Shuffle esto no es posible, ya que habrá intercambio de datos entre particiones y tendrá que hacer escritura en disco.

2.3.1. Evaluación Perezosa (Lazy) La evaluación perezosa significa que Spark esperará hasta el último momento para ejecutar el grafo de instrucciones de cálculo. En lugar de ir modificando los datos inmediatamente después de cada operación, elabora un esquema de transformaciones que deberá aplicar. En el momento que vaya a ejecutarlo tendrá todo el proceso optimizado de la manera más eficiente posible.

Podemos pensar en una serie de transformaciones en las que en el último paso filtramos una fila del DataFrame. Spark al analizar y optimizar las diferentes transformaciones, filtrará esa fila al principio, haciendo que trabajemos con menos datos y de una forma más rápida y eficiente. Spark hará esto de forma autónoma y transparente a nosotros.

1.4 2.4. Acciones:

Mientras que las transformaciones son las que generan el esquema lógico, las acciones son las que desencadenan que Spark calcule un cierto resultado ejecutando todas las transformaciones definidas anteriormente.

Por ejemplo podemos contar el número de divisores de 2 en nuestro DataFrame:

[5]:
```
divisoresDe2.count()
```

[5]:
```
res1: Long = 500
```

Ahora que hemos efectuado una acción, como en este caso count(), es cuando realmente se van a ejecutar todas las transformaciones anteriores. Por supuesto, count() no es la única acción. Existen tres tipos de acciones:

- Acciones para ver datos en la consola.
- Acciones para recopilar datos en objetos nativos en el respectivo lenguaje.
- Acciones para escribir en fuentes de datos externas.

Al ejecutar la acción, iniciamos el job de Spark que ejecuta nuestra transformación de filtro where (una transformación de tipo Narrow), luego una agregación (una transformación de tipo Shuffle) que realiza el recuento en cada de partición, y luego una recopilación, que lleva nuestro resultado a un objeto nativo en el respectivo idioma. Todo esto se puede ver en la Spark UI, que es una herramienta de supervisión de los trabajos que se ejecutan en el cluster. A esta herramienta se accede a través de http://localhost:4040 por defecto cuando trabajamos de forma local.

NOTA: Si creamos varias SparkSession se irán creando en puertos sucesivos a partir del 4040.

1.5 2.5. Ejemplo General:

En los ejemplos anteriores creamos un dataframe de un rango de números, esto realmente no es un trabajo de Big Data. En esta sección reforzaremos los conceptos aprendidos en el tema con un ejemplo más realista y explicaremos paso por paso que es lo que ocurre realmente. Usaremos un archivo sobre datos de la oficina de estadística de transportes de los Estados Unidos. Utilizaremos el archivo 2015-summary.csv que se encuentra en la carpeta Datasets.

Los archivos CSV son un tipo de archivos de datos semiestructurados donde cada fila en el CSV representara una fila de nuestro DataFrame. Spark tiene la habilidad de leer y escribir en un diverso tipo de fuentes de datos. En este caso para leer el archivo haremos:

[6]:
```
val flightData2015 = spark.read
                        .option("inferSchema", "true")
                        .option("header", "true")
                        .csv("./Datasets/2015-summary.csv")
```

[6]:
```
flightData2015: org.apache.spark.sql.DataFrame = [DEST_COUNTRY_NAME: string,
ORIGIN_COUNTRY_NAME: string … 1 more field]
```

En el código podemos ver como en la primera opción *inferSchema* lo que hacemos es lo que se llama inferir esquema, que no es otra cosa que decirle a Spark que intente saber que tipo de dato tiene cada columna. En la segunda opción *header* le decimos a Spark que la primera fila se trata del nombre de las columnas y no de datos.

Este DataFrame inicial, realmente tiene un conjunto de columnas con un número indeterminado de filas. La razón por la que no se especifica el número de filas es porque la lectura de datos es una transformación, y como ya hemos comentado se trata de una operación perezosa. Spark lee solo un par de filas de datos para intentar adivinar de que tipo debería ser cada columna. De todas formas, lo más recomendable sería indicarle de forma explícita el esquema de datos, lo que veremos más adelante.

Para tener una imagen mental del proceso que acabamos de llevar a cabo, observemos la siguiente figura:

Como se puede observar, leemos el archivo CSV como un DataFrame y luego se convierte en una matriz de filas:

[7]:
```
flightData2015.take(3)
```

[7]:
```
res2: Array[org.apache.spark.sql.Row] = Array([United States,Romania,15],
[United States,Croatia,1], [United States,Ireland,344])
```

Ahora, ordenaremos por medio de *sort* nuestro DataFrame respecto a la columna "count" que es de tipo entero.

NOTA: Ten en cuenta que el ordenamiento de un DataFrame no modifica el DataFrame de partida.

Ordenar se trata de una transformación que devuelve un nuevo DataFrame a partir de transformar el anterior. Visualmente:

No sucede nada con los datos cuando ordenamos porque es solo una transformación. Sin embargo, podemos ver que Spark está elaborando un plan sobre cómo ejecutará esto en el cluster. Podemos utilizar la función *explain* en cualquier objeto Dataframe para ver como Spark ejecutará el código:

[8]:
```
flightData2015.sort("count").explain()
```

```
== Physical Plan ==
*(1) Sort [count#28 ASC NULLS FIRST], true, 0
+- Exchange rangepartitioning(count#28 ASC NULLS FIRST, 200), true, [id=#57]
   +- FileScan csv [DEST_COUNTRY_NAME#26,ORIGIN_COUNTRY_NAME#27,count#28]
Batched: false, DataFilters: [], Format: CSV, Location:
InMemoryFileIndex[file:/C:/Users/Gr4vi7y/Datasets/2015-summary.csv],
PartitionFilters: [], PushedFilters: [], ReadSchema:
struct<DEST_COUNTRY_NAME:string,ORIGIN_COUNTRY_NAME:string,count:int>
```

La parte superior es el resultado final y la parte inferior la fuente de datos. En este caso concreto, podemos ver las palabras clave de cada proceso, "Sort", "Exchange" y "Filescan". Esto se debe a que el ordenamiento es una transformación de tipo **Shuffle**, ya que cada dato se debe comparar con el resto para la ordenación. Ahora mismo no es importante entender todo esto, ya que se trata de una herramienta útil para la depuración y mejorar en el conocimiento a medida que se avanza en la formacion en Spark.

Ahora, podemos ejecutar una acción para poner en marcha este plan. Sin embargo, antes vamos a hacer una configuración. De forma predeterminada, cuando realizamos un **shuffle**, Spark hace 200 particiones en orden aleatorio. Vamos a cambiar este valor a 5 para reducir el número de particiones de salida.

[9]:
```
spark.conf.set("spark.sql.shuffle.partitions", "5")
```

[10]:
```
flightData2015.sort("count").explain()
```

```
== Physical Plan ==
*(1) Sort [count#28 ASC NULLS FIRST], true, 0
+- Exchange rangepartitioning(count#28 ASC NULLS FIRST, 5), true, [id=#69]
   +- FileScan csv [DEST_COUNTRY_NAME#26,ORIGIN_COUNTRY_NAME#27,count#28]
Batched: false, DataFilters: [], Format: CSV, Location:
InMemoryFileIndex[file:/C:/Users/Gr4vi7y/Datasets/2015-summary.csv],
PartitionFilters: [], PushedFilters: [], ReadSchema:
struct<DEST_COUNTRY_NAME:string,ORIGIN_COUNTRY_NAME:string,count:int>
```

Repitiendo el código anterior podemos comprobar como en "Exchange", donde se indica el número de particiones, el segundo argumento que antes era 200 ahora es 5. Esto solo se indica como nota para que vayamos entendiendo la diferente información que nos muestra "explain".

Ahora le diremos que nos muestre los 2 primeros valores:

[11]:
```
flightData2015.sort("count").take(3)
```

[11]:
```
res6: Array[org.apache.spark.sql.Row] = Array([Moldova,United States,1], [United
States,Croatia,1], [United States,Singapore,1])
```

Este proceso se ilustra en la siguiente figura:

Spark genera un plan de transformaciones de Dataframes de manera que en cualquier momento puede volver a un DataFrame anterior simplemente realizando los cálculos necesarios a partir del DataFrame de partida. Esto es parte del corazón de Spark, donde no modificamos los datos de forma física, sino que solo creamos un "mapa" de las transformaciones en cada paso. Un ejemplo de esto es la selección del número de particiones, que por defecto eran 200 y hemos configurado en 5. Si jugamos con el número de particiones y analizamos el tiempo de ejecución, veremos que varía drásticamente. Todo esto lo podemos analizar de forma detallada a través del Spark UI (http://localhost:4040).

2.5.1. DataFrames y SQL: En el ejemplo anterior solo vimos un ejemplo sencillo. Ahora haremos un ejemplo más complejo tratándolo como un DataFrame y como SQL puro. Spark puede trabajar de la misma forma independientemente del idioma. Podemos hacer la programación como DataFrames (ya sea con R, Python, Scala o Java) o como SQL puro. Spark compilará la lógica de la misma forma independientemente del lenguaje antes de ejecutar el código. Con Spark SQL podemos hacer una vista (o tabla temporal) y consultarla usando SQL puro. Además, no habrá diferencia en rendimiento lo hagamos de una u otra forma, ya que ambos se "compilarán" de la misma forma en el diseño del "mapa" de ejecución.

Podemos convertir cualquier DataFrame en una tabla o vista con el siguiente método:

[12]:
```
flightData2015.createOrReplaceTempView("flight_data_2015")
```

Ahora podemos hacer cualquier consulta en SQL. Para ello usaremos la función spark.sql (recuerda que spark es nuestra variable SparkSession) que devolverá un DataFrame. Aunque puede parecer un método circular, ya que una consulta en SQL devuelve un Dataframe; en realidad, es bastante poderoso. Esto hace que sea posible especificar las transformaciones de la manera más conveniente sin sacrificar la eficiencia. Para entender esto observemos:

[13]:
```
val sqlWay = spark.sql("""SELECT DEST_COUNTRY_NAME, count(1)
                          FROM flight_data_2015
                          GROUP BY DEST_COUNTRY_NAME""")

val dataFrameWay = flightData2015.groupBy("DEST_COUNTRY_NAME")
                                 .count()

sqlWay.explain()
dataFrameWay.explain()
```

```
== Physical Plan ==
*(2) HashAggregate(keys=[DEST_COUNTRY_NAME#26], functions=[count(1)])
```

```
+- Exchange hashpartitioning(DEST_COUNTRY_NAME#26, 5), true, [id=#98]
   +- *(1) HashAggregate(keys=[DEST_COUNTRY_NAME#26],
functions=[partial_count(1)])
      +- FileScan csv [DEST_COUNTRY_NAME#26] Batched: false, DataFilters: [],
Format: CSV, Location:
InMemoryFileIndex[file:/C:/Users/Gr4vi7y/Datasets/2015-summary.csv],
PartitionFilters: [], PushedFilters: [], ReadSchema:
struct<DEST_COUNTRY_NAME:string>


== Physical Plan ==
*(2) HashAggregate(keys=[DEST_COUNTRY_NAME#26], functions=[count(1)])
+- Exchange hashpartitioning(DEST_COUNTRY_NAME#26, 5), true, [id=#117]
   +- *(1) HashAggregate(keys=[DEST_COUNTRY_NAME#26],
functions=[partial_count(1)])
      +- FileScan csv [DEST_COUNTRY_NAME#26] Batched: false, DataFilters: [],
Format: CSV, Location:
InMemoryFileIndex[file:/C:/Users/Gr4vi7y/Datasets/2015-summary.csv],
PartitionFilters: [], PushedFilters: [], ReadSchema:
struct<DEST_COUNTRY_NAME:string>
```

[13]:
```
sqlWay: org.apache.spark.sql.DataFrame = [DEST_COUNTRY_NAME: string, count(1):
bigint]
dataFrameWay: org.apache.spark.sql.DataFrame = [DEST_COUNTRY_NAME: string,
count: bigint]
```

Como podemos observar, ambos códigos producen exactamente el mismo "mapa" de ejecución.

Ahora usaremos la función max, para calcular el número máximo de vuelos hacia o desde cualquier ubicación. Esto lo que hará será analizar cada valor de la columna relevante en el DataFrame y verificar si ese valor es mayor que los anteriores. Esto es una transformación, ya que solo estamos filtrando una fila.

[14]:
```
spark.sql("SELECT max(count) from flight_data_2015").take(1)
```

[14]:
```
res9: Array[org.apache.spark.sql.Row] = Array([370002])
```

Como era de esperar, tanto de una u otra manera el resultado es el mismo. Ahora buscaremos los cinco principales países de destino. Esta será la primera consulta con varias transformaciones que haremos paso a paso. Empezaremos con SQL:

[15]:
```
val maxSql = spark.sql("""SELECT DEST_COUNTRY_NAME, sum(count) as␣
↪destination_total
                      FROM flight_data_2015
                      GROUP BY DEST_COUNTRY_NAME
```

```
                    ORDER BY sum(count) DESC
                    LIMIT 5""")

maxSql.show()
```

```
+-----------------+-----------------+
|DEST_COUNTRY_NAME|destination_total|
+-----------------+-----------------+
|    United States|           411352|
|           Canada|             8399|
|           Mexico|             7140|
|   United Kingdom|             2025|
|            Japan|             1548|
+-----------------+-----------------+
```

[15]:
```
maxSql: org.apache.spark.sql.DataFrame = [DEST_COUNTRY_NAME: string,
destination_total: bigint]
```

Ahora lo haremos por medio de la sintaxis de DataFrame que es semánticamente similar pero ligeramente diferente en implementación y ordenación. Pero, como ya hemos comentado, los planes subyacentes para ambos son iguales.

[16]:
```
import org.apache.spark.sql.functions.desc

flightData2015.groupBy("DEST_COUNTRY_NAME")
              .sum("count")
              .withColumnRenamed("sum(count)", "destination_total")
              .sort(desc("destination_total"))
              .limit(5)
              .show()
```

```
+-----------------+-----------------+
|DEST_COUNTRY_NAME|destination_total|
+-----------------+-----------------+
|    United States|           411352|
|           Canada|             8399|
|           Mexico|             7140|
|   United Kingdom|             2025|
|            Japan|             1548|
+-----------------+-----------------+
```

[16]:
```
import org.apache.spark.sql.functions.desc
```

Este código tiene siete pasos que nos llevan de regreso a los datos de origen.

El verdadero plan de ejecución (el que veríamos si hacemos "explain") será diferente del que mostramos en la figura. Esto se debe a las optimizaciones que realiza Spark. Estos planos de ejecución es lo que se llama DAG (Gráfico Acíclico Dirigido) de transformaciones, donde en el momento que ejecutamos una acción se genera el resultado.

El primer paso que observamos es la lectura de los datos. Definimos el DataFrame anteriormente, pero como recordatorio, Spark en realidad no lee los datos hasta que se invoca una acción en ese DataFrame o DataFrame derivado del original. El segundo paso es nuestra agrupación; técnicamente cuando llamamos a groupBy, terminamos con un RelationalGroupedDataset, que es una forma elegante de llamar a un DataFrame que tiene una agrupación especifica. Nosotros básicamente especificamos que vamos a agrupar por una clave (o un conjunto de claves) y que vamos a realizar una agregación sobre cada una de esas claves.

Por lo tanto, el tercer paso es especificar la agregación. En este caso usamos el método de agregación suma. Esto toma como entrada una columna o, simplemente, un nombre de columna. El resultado de la suma es un nuevo DataFrame. Es importante recordar (otra vez) que no se ha realizado ningún cálculo aun. Esta es simplemente otra transformación que hemos expresado, y Spark simplemente la añade al plan de ejecución.

El cuarto paso es un simple cambio de nombre. Usamos el método withColumnRenamed que toma dos argumentos, el nombre de la columna original y el nombre de la nueva columna. Por supuesto, esto es otra transformación sin ejecución de ningún tipo de cálculo.

El quinto paso ordena los datos de tal manera que si tuviéramos que tomar los resultados de la parte superior del DataFrame, tendrían los valores más grandes en la columna destination_total.

Como has podido apreciar, hemos tenido que importar la función desc para hacer esto. Y como se ha podido observar, desc no devuelve un string sino una columna. En general muchos métodos de DataFrames aceptarán strings (como los nombres de las columnas), tipos de columnas o expresiones. Realmente columnas y expresiones son lo mismo.

En el penúltimo paso, especificaremos un límite. Esto determina que solo queremos devolver los cinco primeros valores de nuestro DataFrame final, en lugar de todos los datos.

El último paso es nuestra acción. Ahora es cuando empieza el proceso de recolección de los resultados de nuestro DataFrame, y Spark nos dará una lista o array en el lenguaje en el que lo estemos ejecutando. Para repasar todo esto haremos un explain:

[17]:
```
flightData2015.groupBy("DEST_COUNTRY_NAME")
              .sum("count")
              .withColumnRenamed("sum(count)", "destination_total")
              .sort(desc("destination_total"))
              .limit(5)
              .explain()
```

```
== Physical Plan ==
TakeOrderedAndProject(limit=5, orderBy=[destination_total#118L DESC NULLS LAST],
output=[DEST_COUNTRY_NAME#26,destination_total#118L])
+- *(2) HashAggregate(keys=[DEST_COUNTRY_NAME#26], functions=[sum(cast(count#28
as bigint))])
   +- Exchange hashpartitioning(DEST_COUNTRY_NAME#26, 5), true, [id=#240]
```

```
      +- *(1) HashAggregate(keys=[DEST_COUNTRY_NAME#26],
functions=[partial_sum(cast(count#28 as bigint))])
         +- FileScan csv [DEST_COUNTRY_NAME#26,count#28] Batched: false,
DataFilters: [], Format: CSV, Location:
InMemoryFileIndex[file:/C:/Users/Gr4vi7y/Datasets/2015-summary.csv],
PartitionFilters: [], PushedFilters: [], ReadSchema:
struct<DEST_COUNTRY_NAME:string,count:int>
```

Aunque el resultado del explain no coincide con el "plan conceptual" de forma exacta, todos los pasos están realmente ahí. En la primera línea podemos observar el límite de 5 y el ordenamiento por medio de orderBy. También se puede observar que la agregación se realiza en dos pasos, en las llamadas sumas parciales. Esto se debe a que sumar una lista de números tiene la propiedad conmutativa y por lo tanto, Spark puede realizar la suma partición a partición. Y también en último lugar se puede ver la lectura del archivo CSV.

Además de mostrar los datos, también los podemos guardar en un archivo que Spark soporte. Por ejemplo, podríamos querer guardarlo en una base de datos PostgreSQL o en otro tipo de archivo.

1.6 2.6. Conclusión:

En este capítulo introducimos la base de Apache Spark. Hemos hablado sobre las transformaciones, acciones y de como Spark ejecuta de forma perezosa un DAG de transformaciones en un cierto orden para optimizar el plan de ejecución. También hemos explicado como los datos se organizan en particiones y se establece el escenario para transformaciones más complejas.

[ ]: