Ingeniería para el Procesado Masivo de Datos

Tema 4. Spark II

Índice

Esquema

4.1. Introducción y objetivos

En el tema anterior, estudiamos los conceptos básicos de una nueva tecnología de procesado de grandes cantidades de datos en clúster de equipos, denominada Apache Spark. Este *framework* de programación distribuida nacía con la intención de mejorar en términos de eficiencia, rapidez y programación intuitiva el paradigma de MapReduce.

Tal y como se comentaba en el capítulo previo, Spark es un conjunto de componentes, de los que vimos cómo funciona Spark Core y su estructura de datos principal, los RDD. Otro de los componentes, probablemente el más usado hoy en día, es la API estructurada y su estructura de datos clave, los DataFrames. En este capítulo, estudiaremos en detalle esta API estructurada y la creación y manipulación de DataFrames, así como las ventajas que tiene sobre los RDD. Por último, veremos una API relacionada, Spark SQL, que nos facilita la manipulación de estos DataFrames para aquellos desarrolladores con experiencia previa en SQL.

Teniendo en cuenta todo esto, los objetivos que persigue este tema son:

- Conocer la API estructruda de Spark y su principal estructura de datos, el DataFrame.
- Identificar las ventajas de usar DataFrames en lugar de RDD.
- Conocer Spark SQL, así como sus similitudes y diferencias con la API estructurada.
- Practicar con algunas funciones típicas de procesamiento de DataFrames, tanto con la API estructurada como con Spark SQL.

4.2. DataFrames en Spark

Manejar RDD resulta tedioso cuando los tipos de datos que contienen empiezan a complicarse, ya que en todo momento necesitamos saber exactamente la estructura del dato, incluso cuando son tuplas jerárquicas (tuplas en las que algún campo es, a su vez, una tupla o lista). Sería más conveniente poder manejar los RDD como si fuesen tablas de datos, estructuradas en filas y columnas, lo cual aportaría mayor nivel de abstracción y más facilidad de uso. Esto es lo que nos proporcionan la API estructurada y los DataFrames de Spark:

> Un DataFrame de Spark es una tabla de datos distribuida en la RAM de los nodos, formada por filas y columnas con nombre y tipo (incluidos tipos complejos), similar a una tabla en una base de datos relacional

Internamente, un DataFrame no es más que un **RDD de objetos de tipo** `Row` **de Spark**, cada uno de los cuales representa una fila de una tabla como un vector cuyos componentes (lo que serían las columnas de una tabla) tienen nombre y tipo predefinido. El esquema (schema) de un DataFrame define el nombre y el tipo de dato de cada una de estas columnas. **Cada DataFrame envuelve (tiene dentro) un RDD**, al que se puede acceder como el atributo `rdd`. Ej.: `variableDF.rdd`. Por eso, los conceptos de transformación y acción se aplican también a DataFrames.

Hay que recordar que los DataFrame de Spark están distribuidos en la memoria RAM de los nodos *worker*, aunque puedan parecerse a una tabla de una base de datos. Por otro lado, el nombre DataFrame es el mismo que el definido en otras librerías de lenguajes como Python (DataFrames del paquete Pandas) o R (data.frame). Si bien el concepto es el mismo (tabla de datos cuyas columnas tienen nombre y tipo), la implementación y manejo no tienen nada que ver, más allá de que los autores eligieron el mismo nombre. Los DataFrames de Spark son un tipo de dato definido

por Spark, están distribuidos físicamente y se manejan mediante la API de Spark.

Existe, en la API de Spark, un método que permite traerse todo el contenido a una sola máquina (el *driver*) y que devuelve un DataFrame de Pandas (no distribuido). Es el método `toPandas`, pero debe usarse con cuidado, ya que, de nuevo, requiere que todo el contenido distribuido en los nodos quepa en la memoria RAM del nodo donde se ejecuta el proceso *driver*. De lo contrario, provocará una *excepción* `OutOfMemory`.

En las siguientes secciones, vamos a ver las diferentes operaciones que se pueden hacer con DataFrames y la API estructurada. Empezaremos por la lectura y escritura de DataFrames y seguiremos por las diferentes transformaciones que se pueden aplicar a los DataFrames leídos.

4.3. API estructurada de Spark: lectura y escritura de DataFrames

Como cabe esperar, lo primero que se necesita para tener un DataFrame son los datos con los que se quiere trabajar y que, por tanto, serán cargados en un DataFrame para su futura manipulación. Spark puede leer información de numerosas fuentes de datos. Para que Spark pueda conectarse a una fuente de datos, debe existir un conector específico que indique cómo obtener datos de esa fuente y convertirlos en un DataFrame. Entre las fuentes de datos más habituales que disponen de dicho conector, podemos encontrar:

- **HDFS**. Spark puede leer de HDFS diversos formatos de archivo: CSV, JSON, Parquet, ORC y texto plano. No obstante, la comunidad de desarrolladores ha proporcionado mecanismos para leer otros tipos de ficheros, como los XML, entre otros. La terminación indicada en el nombre de archivo no informa a Spark de nada, solo puede servir como pista al usuario que vaya a leer el fichero.

- **Amazon S3**. Almacén de objetos distribuido creado por Amazon, de donde Spark también puede leer cualquiera de los formatos de fichero nombrados anteriormente.

- Bases de datos relacionales, mediante conexiones **JDBC u ODBC**. Spark es capaz de leer en paralelo a través de varios *workers* conectados simultáneamente a una base de datos relacional, cada uno de los cuales lee porciones diferentes de una misma tabla. Cada porción va a una partición del resultado. Spark puede enviar una consulta a la base de datos y leer el resultado como DataFrame.

- Conectores para **bases de datos no relacionales**. Los conectores son específicos, desarrollados por la comunidad o por el fabricante (por ejemplo: Cassandra, MongoDB, HBase ElasticSearch…).

- Cola distribuida **Kafka** (que veremos en un tema posterior). Para datos que se van

leyendo de un *buffer*.

- También datos que llegan en ***streaming* a HDFS** (ficheros que se van creando nuevos en tiempo real).

Aunque hemos hablado de fuentes de datos de lectura, en el caso de querer escribir resultados del DataFrame en almacenamiento persistente, ya sea fichero, base de datos o servicio de mensajería, Spark proporciona los mismos mecanismos de los que hemos hablado para la lectura. A continuación, vamos a ver en detalle cómo realizar las lecturas y escrituras de los datos.

Lectura de DataFrames

En general, para leer datos desde Spark, se usa un atributo de la `SparkSession`, `spark.read`, y se especifican diferentes opciones, dependiendo del tipo de fichero por leer:

- **Formato**. Tal y como se mencionó anteriormente, el formato de fichero puede ser CSV, JSON, Parquet, Avro, ORC, JDBC/ODBC o texto plano, entre otros muchos.
- **Esquema**. Hay cuatro opciones referentes al esquema:

1. Algunos tipos de fichero almacenan el esquema junto a los datos (por ejemplo, Parquet, ORC o Avro) y, por tanto, no es necesario indicar ningún esquema adicional.
2. Para los tipos de fichero que no almacenan el esquema, es posible solicitar a Spark que trate de inferirlo con la opción `inferSchema` activada (`true`). Hay que tener en cuenta que esta opción conllevará más tiempo de lectura, dado que Spark necesita leer una serie de líneas del fichero y analizarlas para tratar de adivinar el esquema.
3. Podemos pedir a Spark que no trate de inferir el esquema. En este caso, todos los datos se leerán como si fueran texto (*string*).
4. Cabe la opción de indicar de forma explícita el esquema de los datos esperado, para

evitar un incremento del tiempo de lectura y posibles incorrecciones en la inferencia del esquema por parte de Spark. Veremos estas opciones en detalle, con algún ejemplo, más adelante.

- **Modo de lectura**. Puede ser; permissive (por defecto), que traduce como null aquellos registros que considere corruptos de cada fila; dropMalformed , que descarta las filas que contienen alguno de sus registros con un formato incorrecto, y failFast , que lanza un error en cuanto encuentra un registro con un formato incorrecto.

- Existen otra serie de opciones que veremos más adelante, ya que dependen del tipo específico de fichero a leer.

La única información obligatoria es la ruta del fichero que va a ser leído; el resto de opciones son opcionales. Por tanto, la estructura genérica de lectura sería la siguiente:

```
myDF = spark.read.format(<formato>)
        .load("/path/to/hdfs/file")  # spark es el objeto SparkSession
    # <formato> puede ser “parquet” | “json” | “csv” | “orc” | “avro”
```

En cuanto al esquema, recordemos que es el que describe el nombre y tipo de cada uno de los registros (columnas) de las filas del DataFrame. Como comentábamos, algunos tipos de fichero, como Avro, Parquet u ORC, contienen información respecto a su esquema y, por tanto, no es necesario especificarlo durante su lectura. Sin embargo, con otros formatos, como CSV o JSON, donde el esquema del fichero no está almacenado en este, podemos dejar que Spark infiera el esquema con la opción inferSchema configurada como *true*, indicar que no infiera nada (inferSchema a false) y lea todo como *string*, o bien especificar explícitamente cuál va a ser el esquema concreto esperado.

Cabe recordar una vez más que la opción inferSchema indica a Spark que trate de

adivinar el tipo de cada columna, en cuyo caso Spark tratará de inferir lo mejor posible esta información. No obstante, puede darse el caso de que, en una columna que debería ser de tipo entero, falte alguna información y de que Spark, ante la duda, infiera que dicha columna es de tipo texto (*string*). Para evitar este tipo de incorrecciones, es mejor especificar explícitamente el esquema si se conoce de antemano. Un esquema en Spark no es más que un objecto `StructType`, compuesto por un conjunto de `StructFields`. Cada `StructField` representa un registro (columna) de una fila; por tanto, se compone de un nombre (`name`), un tipo (`type`), un booleano que indica si la columna puede contener datos `null` (es decir, datos inexistentes), así como otra información opcional. Por ejemplo, podemos definir en `pyspark` el esquema de un fichero JSON donde cada fila contiene tres campos (columnas) de la siguiente forma:

```
from pyspark.sql.types import StructField, StructType, stringType, LongType

fileSchema = StructType([
 StructField(“dest_country_name”, StringType(), True),
 StructField(“origin_country_name”, StringType(), True),
 StructField(“count”, LongType(), False)
])
```

Una vez que se tiene el esquema definido, solo resta leer los datos utilizando dicho esquema:

```
myDF = spark.read.format("json")
                        .schema(fileSchema)
                        .load("/path/to/file.json") # spark es el objeto
SparkSession
```

Si no queremos que Spark infiera el esquema ni proporcionar uno nosotros, Spark leerá todas las columnas como *strings*:

```
myDF = spark.read.format(“json”)
 .options(“inferSchema”, “false”)
 .load(“/path/to/file.json”) # spark es el objeto SparkSession
```

En caso de querer que Spark infiera el esquema en lugar de especificarlo, habría que usar la opción `inferSchema`, como comentábamos anteriormente:

```
myDF = spark.read.format("json")
                        .options("inferSchema", "true")
                        .load("/path/to/file.json") # spark es el objeto
SparkSession
```

Por último, cabe mencionar que, para algunos tipos de fichero, suele estar disponible un atajo como `spark.read.<format>(“/path/to/file”)`, donde `format` puede ser, por ejemplo, `cvs` o `avro`, aunque no todos los formatos lo tienen.

Vamos a ver algunas particularidades de ciertos formatos concretos.

CSV

Los ficheros CSV son probablemente los más problemáticos, ya que la división de las filas en diferentes registros (columnas) depende de separadores que no siempre se respetan o que son confusos respecto al texto del fichero. Por ejemplo, si los registros están separados por comas, y uno de ellos es de tipo *string* y puede contener dentro comas, cabe la posibilidad de que se produzcan interpretaciones (*parser*) incorrectas. Además de las opciones que ya hemos visto, los ficheros CSV soportan más opciones. Entre las más usadas, destacan:

- Si el fichero incorpora cabecera, la opción `header` indica si cabe esperar que la primera línea del fichero se corresponda con los nombres de las columnas (`true`) o no (`false` , por defecto).
- El carácter separador. La opción `delimiter` permite indicar el carácter separador de

registros (columnas) en una fila. Es importante tener en cuenta que **Spark no soporta separadores de más de un carácter**.

Veamos un ejemplo de lectura de CSV con y sin inferencia de esquema, en el que usaremos, además, el atajo .csv(<path>) que se comentaba anteriormente.

```
# En df1 se van a leer todas las columnas como si fuesen strings:

df1 = spark.read.option("inferSchema", "false")
                        .csv("/path/hdfs/file")#uso del atajo

# Para evitar que todas las columnas se lean como strings,
# vamos a indicar el esquema para que cada columna
# se lea con su tipo correspondiente y se le asigne el nombre deseado:

myschema = StructType([
                        StructField("columna1", DoubleType(), nullable =
False),
                        StructField("columna2", DateType(), nullable =
False)

])

# Pasamos el esquema para que se lean correctamente
# (el true/false como valor de option se escribe en minúscula):

df2 = spark.read.option("header", "true")\
                        .option("delimiter","|")\
                        .schema(myschema)\
                        .csv("/path/hdfs/file")#uso del atajo
```

Parquet, ORC, Avro

Para estos tipos de ficheros, la lectura es más directa, ya que basta con ejecutar el siguiente código (no es necesario ni inferir esquema ni especificarlo):

```
myDf = spark.read.parquet("/path/to/file.parquet") # usa atajo
```

Escritura de DataFrames

La operación de escritura es análoga a la de lectura. En este caso, se usa el atributo *write* de los DataFrames y se indican los siguientes aspectos:

- **Formato**, con el método format(<formato>) , que puede ser cualquiera de los que hemos visto para la operación de lectura.
- **Modo de escritura**. Puede ser append, overwrite, errorIfExists , *ignore* (si los datos o ficheros existen, no se hace nada).
- Otras opciones específicas de cada formato.

Así, por ejemplo, si queremos guardar un DataFrame, df, en formato CSV, utilizaremos el tabulador como separador y, en modo sobreescritura, en la cabecera en el fichero, escribiremos algo como lo siguiente:

```
df.write.format("csv")
                        .mode("overwrite")
                        .option("sep", "\t")
                        .option("header", "true")
                        .save("path/to/hdfs/directory")
```

Mientras que, si queremos guardarlo en Parquet, introduciremos el siguiente código:

```
df.write.format(“parquet”) # equivalente en orc, avro y json
 .mode(“overwrite”)
 .save("path/to/hdfs/directory")
```

Hay que tener en cuenta que la ruta especificada donde se guardará la información no es un fichero, sino un directorio (que creará Spark), dentro del cual se escribirá la información del DataFrame en diferentes partes, con la estructura part-XXXXX-hash.csv en el caso de CSV. También puede escribirse el resultado en otros destinos de datos muy diversos, siguiendo sintaxis específica. Para más detalles, se puede consultar la web de Spark.

Ideas clave

4.4. API estructurada de Spark: manipulación de DataFrames

Una vez que sabemos qué es un DataFrame, cómo leer datos para crear DataFrames y cómo escribir la información que albergan en almacenamiento persistente, veamos qué tipo de operaciones podemos realizar sobre estas estructuras de datos distribuidas que nos proporciona Spark.

Recordemos que un DataFrame consistía en un conjunto de filas (en el fondo, envolvían RDD de objetos de tipo `Row`), cada una de las cuales estaba formada por una serie de registros (columnas). La mayoría de las operaciones que ofrece la API estructurada son operaciones sobre columnas (clase `Column`). Spark implementa muchas operaciones entre columnas de manera distribuida (operaciones aritméticas entre columnas, manipulación de columnas de tipo *string*, comparaciones…), que debemos usar siempre que sea posible. Todas ellas son sometidas a **optimizaciones por parte de Catalyst** para ejecutar más rápidamente.

Para más información sobre la API estructurada, accede a la documentación oficial: https://spark.apache.org/docs/latest/api/python/

La utilización general de los métodos de la API sigue el siguiente formato: `objetoDataFrame.nombreDelMétodo(argumentos)`. Todas las manipulaciones devuelven como resultado un nuevo DataFrame, sin modificar el original (recordamos que los RDD son **inmutables** y, por extensión, los DataFrames también). Por eso, se suelen encadenar transformaciones: `df.método1(args1).método2(args2)`

Transformaciones más frecuentes con DataFrames

Supongamos que hemos creado una variable llamada `df` con esta línea de código:

```
df = spark.read.parquet("/ruta/hdfs/datos.parquet")
```

Antes de describir las transformaciones más frecuentes con DataFrames, cabe mencionar que todas ellas están contenidas en la librería de pyspark pyspark.sql.functions. Por tanto, es necesario importar esta librería antes de usar cualquiera de estas funciones. Generalmente, se suele importar asignándole el alias F, de forma que, posteriormente, nos podremos referir a las funciones como F.nombreFuncion(argumentos).

Una vez que sabemos cómo importar las funciones, veamos cuáles son:

- printSchema imprime el esquema del DataFrame. Esta función es muy útil cuando queremos comprobar que el DataFrame se ha leído de fichero con el tipo de datos esperado.

```
df.printSchema()
```

- col("nombreCol") sirve para seleccionar una columna y devuelve un objeto Column sobre el que podemos realizar diferentes transformaciones. Es importante tener en cuenta que no se puede mezclar código SQL (que devuelve DataFrames) con objetos de tipo Column.

- select permite seleccionar columnas de diferentes formas:

```
import pyspark.sql.functions as F

#ambas formas de usar select devuelven el mismo resultado:

df.select("nombreColumna").show(5)

df.select(F.col("nombreColumna")).show(5)

# Crear columna con nombre diff y seleccionar esa columna,
# que es el resultado de restar el literal 18 a la columna edad.
```

```
# Mostrar 5 registros de la columna resultante seleccionada:

df.select((F.col("edad")-F.lit(18)).alias("diff")).show(5)
```

- alias le asigna un nombre a la columna sobre la que se aplica

```
import pyspark.sql.functions as F

df.select(F.col("nombreColumna").alias("nombreNuevo")).show(5)
```

- withColumn devuelve un nuevo DF con todas las columnas del original más una nueva columna añadida al final, como resultado de una operación entre columnas existentes que devuelve como resultado un objeto Column

```
import pyspark.sql.functions as F

# Crea una nueva columna cuyo nombre es nombreNuevaColumna

# y que almacena la suma de los valores en las columnas c1 y c2.

# Devuelve un objeto de tipo Column, del que mostramos 5 registros:

df.withColumn("nombreNuevaColumna", F.col("c1")+F.col("c2")).show(5)
```

- drop elimina una columna.

```
import pyspark.sql.functions as F

# Ambas formas de utilizar drop dan el mismo resultado

df.drop("nombreColumna")

df.drop(F.col("nombreOtraColumna"))
```

- withColumnRenamed renombra una columna.

```

```

```
import pyspark.sql.functions as F

df.withColumnRenamed("nombreExistenteColumna", "nombreNuevoColumna")
```

- when(condición, valorReemplazo1).otherwise(valorReemplazo2) sirve para reemplazar valores de una columna según una condición que implica a esa o a otras columnas. Si no especificamos otherwise , los campos donde no se cumpla la condición se rellenarán a null. Esta función se utiliza generalmente dentro de withColumn:

```
import pyspark.sql.functions as F

df.withColumn("esMayor",F.when("edad>18", "mayor").otherwise("menor"))
```

Transformaciones matemáticas y estadísticas con DataFrames

Existen numerosas funciones matemáticas y estadísticas disponibles para su aplicación sobre DataFrames. Dichas funciones generan columnas y DataFrames nuevos como, por ejemplo, los siguientes:

- Columna de números aleatorios:

```
import pyspark.sql.functions as F

# Crea una nueva columna con números aleatorios de una uniforme:
df = df.withColumn("unif", F.rand())

# Crea una nueva columna con números aleatorios de una normal:
df = df.withColumn("norm", F.randn())
```

- DataFrame con estadísticos descriptivos (media, desviación típica, máximo, mínimo…) (método *describe*):

```
df.describe().show()
```

- Operaciones matemáticas, entre otras:
  - Seno: F.sin())
  - Coseno: F.cos(F.col("nombreColumna"))
  - Raíz cuadrada: F.sqrt(F.col("nombreColumna"))

Combinaciones y filtrado de DataFrames

Otro grupo de funciones son las relacionadas con la combinación y filtrado de DataFrames. Entre ellas, podemos encontrar:

- Unión de DataFrames

```
df3 = df1.unionAll(df2)
```

- Diferencia de DataFrames:

```
df3 = df1.except(df2)
```

- Filtrado de filas de un DataFrame. Existen dos formas de expresar la condición de filtrado:

```
import pyspark.sql.functions as F

# df3 tendrá las filas de df donde la columna c1 tenga un valor
# mayor que c2
df3 = df.where(F.col("c1")>F.col("c2"))

df3 = df.where("c1 > c2") # Es equivalente a lo anterior
```

Operaciones de RDD aplicadas a DataFrames

En general, las operaciones que podíamos usar sobre RDD suelen estar disponibles

también para los DataFrames. No olvidemos que un DataFrame no es más que un RDD de objetos de tipo `Row` y que podemos acceder al RDD que envuelve el DataFrame mediante el atributo `rdd`:

```
df.rdd.take(5) # Muestra 5 objetos de tipo Row
```

- `map` y `flatmap` sobre el RDD; en este caso, iteramos sobre objetos de tipo `Row`:

```
<pr><code>def mifuncion(r): # r es un Row y se accede a sus campos mediante
'.'

return((r.DNI, "Bienvenido " + r.nombre + " " + r.apellido))

paresRDD = df.rdd.map(mifuncion) # map sobre un RDD devuelve un RDD

dfRenombrado = paresRDD.toDF("DNI", "mensaje") # lo convertimos a DF
```

- Otras operaciones que pueden realizarse son: transformaciones como como `sample`, `sort`, `distinct`, `groupBy` …, y acciones como `count`, `take`, `first` , etc.

A continuación, se muestra un ejemplo un poco más completo del uso de la API estructurada:

```
# el objeto spark (sparkSession) ya está creado en Jupyter. Si no,
# habría que crearlo indicando la dirección IP del máster
# de un cluster de Spark existente:
from pyspark.sql import functions as F
df = spark.read.parquet("/mis/datos/en/hdfs/flights.parquet")
resultDF = df.withColumn("distMetros", F.col("dist")*1000)\
             .withColumn("retrasoCat",
F.when(F.col("retraso") < 15, "poco")\
  .when(F.col("retraso") < 30, "medio")
  .otherwise("mucho"))  # sin otherwise, pondría Null!
  .select(F.col("aeronave"), F.col("origen"),
   F.col("disMetros"),
   F.col("retrasoCat"),
   (F.col("retraso") / F.col("dist")).alias("retrasoPorKm")
 )\
```

```
   .withColumnRenamed("dist", "distKm")
   .where(F.col("aeronave") == "Boing 747") # equivale a .filter()
   .where("distMetros > 20000 and origen != 'Madrid'")

# Línea anterior: where pasa una consulta SQL como string.
# En la línea siguiente, hacemos de nuevo lo mismo:
df2 = resultDF.select("retraso, compania")
       .where("aeropuerto like '%Barajas' and retrasoCat = 'poco'")
df2.show() # show es una acción que provoca que se calculen todos los
 # DataFrames de las transformaciones anteriores hasta show.
```

Operaciones de agrupamiento y agregación

El método groupBy("nombreCol1", "nombreCol2" , …) sobre un DataFrame devuelve una estructura de datos llamada RelationalGroupedDataset , que no es un DataFrame y sobre la que apenas se pueden aplicar operaciones. Equivale a la operación GROUP BY de SQL, donde se definen grupos para después calcular, para cada uno, un resultado agregado (por ejemplo, la suma, la media, un conteo, etc.) de una o varias variables numéricas.

Se suelen aplicar operaciones como count() , que efectúa un conteo del número de elementos de cada grupo, o la función agg() , que es la más habitual y realiza, para cada grupo, las agregaciones que le indiquemos sobre las columnas seleccionadas. El resultado solamente contendrá aquellas columnas que se incluyeron como argumentos en el groupBy más aquellas que sean mencionadas como argumento de alguna de las operaciones de agregación que incluimos en la función agg.

Cuando ejecutamos funciones de agregación sin haber llevado previamente una agrupación con groupBy , lo que obtenemos es un DataFrame con una sola fila, que es el resultado de la agregación. El fragmento de código siguiente muestra cómo se utilizan groupBy y agg.

```
import pyspark.sql.functions as F
newDF = myDF.agg(max(F.col("mycol"))) # devuelve DF de una sola fila
newDF = myDF.groupBy("mycol").agg(F.max(F.col("mycol"))) #Tantas filas
                                  # como  valores distintos en mycol
```

```
newDF = myDF.withColumn("complicated", # F.sin: seno. F.lit: constante
      F.lit(2)*F.sin(F.col("colA"))*F.sqrt(F.col("colB")))
newDF = myDF.groupBy("id").agg(F.count("id").alias("countId"),
      F.max("date").alias("maxdate"),
 F.countDistinct("prod").alias("nProd"))
# Sintaxis tipo diccionario para indicar varias agregaciones:
newDF = myDF.groupBy(F.col("someCol"))
      .agg({"existingCol": "min", "otherCol": "avg"})
# Indicamos que, en cada grupo, definido por cada valor de "someCol",
# queremos el mínimo de la columna "existingCol" en dicho grupo y la
# media de los valores de la columna "otherCol". Esto devuelve
# un DF con tantas filas como valores distintos tenga someCol.
```

4.5. Ejemplo de uso de API estructurada

En el capítulo anterior, vimos un ejemplo completo de manejo de RDD, con la intención de replicarlo con la API estructurada para observar las diferencias de uso entre ambas API.

Recordemos que, para ello, usábamos los ficheros simplificados flights.csv y airport-codes.csv, almacenados ambos en HDFS, bajo el directorio /user/data, y cuyas primeras líneas son las siguientes:

```
“year”,”month”,”day”,”origin”,”dest”
2014,1,1,”PDX”,”ANC”
2014,1,1,”SEA”,”CLT”
…
```

```
ident,name,iso_country,iata_code
LELT,Lillo,ES,
LEMD,Adolfo Suárez Madrid–Barajas Airport,ES,MAD
…
```

El objetivo del ejemplo era contar cuántos vuelos reciben los diferentes destinos (dest) que aparecen en el fichero flights.csv. Además, como no conocemos bien los códigos de los aeropuertos de llegada, queríamos ver esta agregación como nombre completo del aeropuerto y número de vuelos que recibe. Para esto, nos ayudábamos de la información que contiene el fichero arirport-codes.csv, donde se relacionaba el código del aeropuerto con su nombre. Veamos entonces cómo realizar esta misma funcionalidad usando la API estructurada.

```
# Empezamos obteniendo la SparkSession.
# Como, en este caso, no vamos a usar RDD, no es necesario
# obtener el SparkContext:
from pyspark.sql import SparkSession
```

```
import pyspark.sql.functions as F

spark = SparkSession.builder.appName("ejemplo_DF")\
    .getOrCreate()

# Lo siguiente es cargar los datos de fichero
# en un DataFrame, usando las funciones de lectura
# que vimos anteriormente.
# Concretamente, usamos la opción header, que nos
# permite no tener que hacer código auxiliar para no cargar
# la cabecera como datos, como pasaba con RDD:
flightsDF = spark.read\
            .option("header", "true")\
            .csv("hdfs://<ip:port>/user/data/flights.csv")

# Cuando usamos DataFrames, tenemos dos opciones para obtener
# registros de ellos:
# Usar la función take de RDD:
print(flightsDF.take(5))

# Usar la función show de DataFrames:
print(flightsDF.show(5))
```

Vemos que, cuando usamos la acción `take`, esta devuelve cinco registros del RDD envuelto por el DataFrame. Y recordemos que un DataFrame no era más que una envoltura de un RDD de objetos de tipo `Row`, tal y como podemos comprobar con el resultado de mostrar los registros devueltos por `take`. Por otro lado, la API estructurada nos proporciona un método mucho más visual como es show, que muestra la misma información, pero en formato tabla.

```
[Row(year='2014', month='1', day='1', origin='PDX', dest='ANC'),
 Row(year='2014', month='1', day='1', origin='SEA', dest='CLT'),
 Row(year='2014', month='1', day='1', origin='PDX', dest='IAH'),
 Row(year='2014', month='1', day='1', origin='PDX', dest='CLT'),
 Row(year='2014', month='1', day='1', origin='SEA', dest='ANC')]

+----+-----+---+------+----+
|year|month|day|origin|dest|
+----+-----+---+------+----+
|2014|    1|  1|   PDX| ANC|
|2014|    1|  1|   SEA| CLT|
|2014|    1|  1|   PDX| IAH|
```

```
|2014|    1|  1|   PDX| CLT|
|2014|    1|  1|   SEA| ANC|
+----+-----+---+------+----+
only showing top 5 rows
```

```
# Otro método útil que nos proporciona la API estructurada
# es printSchema, para comprobar el tipo de datos de cada
# columna del DataFrame:
flightsDF.printSchema()
```

Con printSchema, podemos comprobar el tipo de datos leído por Spark. Como, en este caso, no hemos indicado ni esquema ni valor para inferSchema, se aplica por defecto *false* y se lee todo como *string*.

```
root
 |-- year: string (nullable = true)
 |-- month: string (nullable = true)
 |-- day: string (nullable = true)
 |-- origin: string (nullable = true)
 |-- dest: string (nullable = true)
```

```
# Esto nos da pie a convertir el tipo de las columnas
# año, mes y día, y ver así cómo se hace.
flightsDF = flightsDF\
            .withColumn('year', F.col('year').cast(IntegerType()))\
            .withColumn('month', F.col('month').cast(IntegerType()))\
            .withColumn('day', F.col('day').cast(IntegerType()))
```

```
# Ahora, para contar cuantos vuelos llegan a cada aeropuerto de
# destino, basta con agrupar por destino y contar registros por grupo:
flights_destDF = flightsDF.groupBy('dest').count()

# Le cambiamos el nombre a la columna count para ver cómo se hace:
flights_destDF = flights_destDF\
                   .withColumnRenamed('count','dest_count')

# Ordenamos el DataFrame por número de vuelos en orden
# descendente y mostramos los 5 primeros:
flights_destDF.orderBy(F.desc('dest_count')).show(5)
```

Como vemos, es mucho más sencillo hacer este tipo de manipulaciones con la API estructurada, ya que nos despreocupamos de usar RDD o pairRDD , de la estructura del RDD, etc., y solo tenemos que prestar atención a la estructura de tabla y a las operaciones que queremos realizar.

```
+----+----------+
|dest|dest_count|
+----+----------+
| SFO|     12809|
| LAX|     10456|
| DEN|      9518|
| PHX|      8660|
| LAS|      8214|
+----+----------+
only showing top 5 rows
```

```
# Para mostrar también el nombre de los aeropuertos,
# vamos a cargar el fichero correspondiente en otro DataFrame:
airportsDF = spark.read\
            .option("header", "true")\
            .csv("hdfs://<ip:port>/user/data/airport-codes.csv")

# Y solo queda hacer el join. Del resultado del join, nos quedamos
# con el nombre del aeropuerto (name) y el número de vuelos,
# y ordenamos por número de vuelos en orden descendente:
flights_airports = flights_destDF\
      .join(airportsDF, flights_destDF.dest == airportsDF.iata_code)\
      .select(F.col('name'), F.col('dest_count'))\
      .orderBy(F.desc('dest_count'))
flights_airports.show(5)
```

Obtenemos los mismos resultados que con RDD, pero de una forma mucho más intuitiva y menos proclive a producir errores.

```
+------------------+----------+
|              name|dest_count|
+------------------+----------+
|San Francisco Int...|     12809|
|Los Angeles Inter...|     10456|
|Denver Internatio...|      9518|
|Phoenix Sky Harbo...|      8660|
```

```
|McCarran Internat...|      8214|
+--------------------+----------+
only showing top 5 rows
```

```
# Por último, para demostrar el uso, escribimos el resultado en ficheros:
flights_airports.write\
   .format('csv')\
   .save("hdfs://192.168.240.4:9000/user/data/flights_dest_airports")
```

Con este ejemplo, comprobamos de primera mano las facilidades que brinda la API estructurada, además de proporcionar las optimizaciones derivadas del uso del motor Catalyst. Queda claro por qué los desarrolladores de Spark recomiendan encarecidamente el uso de esta API, siempre que sea posible, en lugar de la de RDD.

4.6. Spark SQL

Spark SQL ofrece una potente opción, que consiste en aplicar operaciones escritas como consultas en lenguaje SQL a DataFrames que se hayan registrado como tablas, sin tener que utilizar la API estructurada paso a paso. Es decir: Spark SQL se integra con la API de DataFrames; así, se puede expresar parte de una consulta en SQL y parte utilizando la API estructurada. Sea cual sea la opción elegida, cabe recordar que se compilará en el mismo plan de ejecución, dado que, como se veía en la descripción de Spark, tiene un motor de ejecución unificado.

Antes de que Spark fuera una herramienta tan usada como lo es hoy en día, la tecnología *big data* que permitía hacer consultas usando lenguaje SQL sobre grandes conjuntos de datos almacenados de forma distribuida era Hive (que se estudiará en un capítulo posterior). Esta herramienta fue muy popular, ya que ayudó a que Hadoop fuera usado por profesionales que no tenían conocimientos suficientes de Java u otros lenguajes de programación para realizar procesamientos de los datos almacenados en HDFS utilizando MapReduce, pero que sí tenían amplios conocimientos en el uso de bases de datos SQL.

Por su parte, Spark comenzó como un motor de procesamiento paralelo de grandes cantidades de datos basado en RDD. Sin embargo, a partir de la versión 2.0, los autores incluyeron un *parser* de consultas SQL (que soportaba tanto ANSI-SQL como HiveQL, el lenguaje SQL de Hive), que dio lugar a una herramienta similar a Hive. Es decir, ofrecía la posibilidad de realizar consultas SQL sobre un conjunto de datos sin necesidad de tener conocimientos de Python, Java o Scala para ello, lo que hacía más accesible el procesamiento de grandes conjuntos de datos. **Spark SQL no reemplaza a Hive,** pero sí que incluye prácticamente toda la funcionalidad que proporciona Hive, atada a que la ejecución por debajo está anclada al motor de procesamiento de Spark.

Ideas clave

Cabe aclarar que **Spark SQL** está pensado para funcionar como un **motor de procesado de consultas en *batch*** (OLTP) y no para realizar consultas interactivas o que necesiten una baja latencia (OLAP). Exactamente igual ocurre con Hive, como se comentará en el capítulo correspondiente.

Spark SQL proporciona su funcionalidad gracias al uso de un catálogo de metadatos, denominado Catalog. Dicho **Catalog** es una **abstracción del *metastore***, es decir, del almacenamiento de los metadatos que definen las tablas y vistas (*dataframes* registrados) sobre los que ejecuta las consultas SQL. El Catalog envuelve la complejidad del *metastore* y proporciona una serie de funciones que se pueden ejecutar sobre él (por ejemplo, listar bases de datos, tablas y vistas, y funciones).

E l ***metastore*** representado por el Catalog puede almacenarse bien en memoria (opción 'in-memory', por defecto al usar Spark con scripts), o bien en un **metastore** de Hive (opción 'hive', por defecto al usar Spark Shell, como con Jupyter Notebooks). En caso de usar la opción 'hive', se puede utilizar un *metastore* ya existente y que se especifica mediante configuración, o bien lo crea el propio Spark en caso de no especificar ninguno. En caso de usar un *metastore* de Hive. este metastore se puede acceder desde fuera de Spark, por ejemplo desde Hive o herramientas BI, de forma que es posible por ejemplo manejar y procesar datos con Spark, registrar las tablas resultantes en el *metastor*, para luego consultarlas con Hive o herramientas BI, configurando el acceso al mismo *metastore*. De igual forma, en caso de usar un *metastore* ya existente en el que hubiera tablas registradas, Spark SQL puede acceder y modificar dichas tablas aunque no hayan sido creadas desde Spark SQL.

Existen tres opciones para ejecutar consultas en Spark SQL:

- **Interfaz de línea de comandos de Spark SQL**. Es una herramienta útil para realizar consultas sencillas en modo local desde la línea de comandos. Para acceder a esta herramienta, basta con ejecutar `./bin/spark-sql` en línea de comandos, en el directorio donde esté Spark instalado.

- **API de Spark SQL**. El método . sql() de la sparkSession recibe como argumento una consulta SQL, que puede hacer referencia a cualquier tabla registrada en *metastore,* y devuelve un DataFrame con el resultado de la consulta. Para registrar un DataFrame en el *metastore,* lo cual genera (solamente) los metadatos necesarios, existen varios métodos; entre ellos, podemos mencionar createOrReplaceTempView , que la registra solo durante esta sesión, o write.format('hive').saveAsTable("nombre_tabla") , que la registra de forma permanente para futuras sesiones. Spark ofrece más métodos para manejar el *metastore*, pero quedan fuera del alcance de este tema. Spark analiza automáticamente la consulta y la traduce a un DAG, exactamente del mismo modo que ocurre al utilizar la API estructurada que hemos visto en las secciones anteriores. Veamos un ejemplo:

```
from pyspark.sql import functions as F
df = spark.read.parquet("/mis/datos/en/hdfs/flights.parquet")
df.createOrReplaceTempView("vuelos") # crear tabla temporal vuelos
resultDF = spark.sql("select *, 1000*dist as distMetros from vuelos")
resultsDF.show() # todas las cols originales más una nueva distMetros
```

Además, la API SQL es totalmente interoperable con la API estructurada, de forma que se puede crear un DataFrame, manipularlo primero con SQL y después con la API estructurada. Es decir, se puede hacer código como el siguiente:

```
from pyspark.sql import functions as F
df = spark.read.parquet("/mis/datos/en/hdfs/flights.parquet")
df.createOrReplaceTempView("vuelos") # crear tabla temporal vuelos
resultDF = spark.sql("select *, 1000*dist as distMetros from vuelos")
  .where("distMetros > 100000") # API SQL + estructurada
resultsDF.show() # Todos los vuelos con distancias >100 km
```

- **Servidor JDBC/ODBC**. Spark proporciona una interfaz JDBC/ODBC, mediante la cual aplicaciones BI, como Tableau, tienen acceso al *metastore* gestionado por Spark SQL, y pueden lanzar consultas SQL sobre las tablas registradas en dicho *metastore,* que se ejecutarán de forma distribuida sobre el clúster de Spark.

Ideas clave

Tablas en Spark SQL

El elemento de trabajo en Spark SQL son las tablas, equivalente a los DataFrames en la API estructurada. Toda tabla pertenece a una base de datos (*database*) y, si no se especifica ninguna, lo hará a la base de datos por defecto (*default*). Las tablas siempre contienen datos y no existe el concepto de tabla temporal. En su lugar, existen vistas, que no contienen datos. Es importante tener esto en cuenta a la hora de eliminar (drop) vistas (no se eliminan datos) y tablas (se eliminan datos).

Otro aspecto que debemos tener en cuenta al crear tablas es si se desea que estas sean gestionadas por Spark (*managed table*) o no (*unmanaged table*). Para entender este concepto, cabe mencionar que una tabla está formada por dos tipos de información: los datos que contiene y los metadatos que la describen. Con esto presente, podemos ver cómo se diferencian las tablas gestionadas y no gestionadas por Spark:

- **Tablas gestionadas por Spark**. Cuando se guarda un DataFrame como una nueva tabla (usando por ejemplo saveAsTable sobre un DataFrame, o CREATE TABLE), entonces se crea una tabla gestionada por Spark, ya que son datos nuevos que necesitan almacenarse, y Spark es el encargado de ello. De esta forma, Spark es responsable tanto de los datos como de los metadatos de estas tablas gestionadas, y si borramos estas tablas desde Spark, se borrarán tanto los datos como los metadatos.

- **Tablas no gestionadas por Spark**. En este caso, Spark gestiona solo los metadatos asociados a la tabla, mientras que nosotros gestionamos los datos, es decir, nosotros controlados dónde se guardan y cuándo se borran. Por tanto, cuando borremos la tabla no gestionada con Spark, solo se eliminarán los metadatos, mientras que los datos quedarán almacenados hasta que nosotros decidamos borrarlos. Podemos crear tablas no gestionadas por Spark de diversas formas. Por ejemplo, cuando se define una tabla desde ficheros ya almacenados en disco, lo que se crea es una tabla no gestionada por Spark, dado que los datos ya existían

previamente, no son datos nuevos creados usando Spark. Esto se consigue especificando un path concreto antes de usar `saveAsTable` (por ejemplo, `df.write.option("path", "/path/to/save").saveAsTable("table_name")`) . Otra opción es utilizar `CREATE EXTERNAL TABLE` , lo que se conoce en muchos foros como tablas externas. Esta opción existe por compatibilidad con Hive, y permite crear tablas no gestionadas por Spark. Para crear una tabla externa se utiliza la consulta `CREATE EXTERNAL TABLE` , tal y como se muestra en el ejemplo a continuación:

```
CREATE EXTERNAL TABLE flights (
 DEST_COUNTRY_NAME STRING, ORIGIN_COUNTRY_NAME STRING)
ROW FORMAT DELIMITED
 FIELDS TERMINATED BY ‘,’
LOCATION ‘data/flight_info/’
```

O también desde el resultado de otra tabla:

```
CREATE EXTERNAL TABLE flights
ROW FORMAT DELIMITED
 FIELDS TERMINATED BY ‘,’
LOCATION ‘data/flight_info/’
AS SELECT * FROM flights
```

Cuando queremos eliminar una tabla, usamos la consulta DROPo. Es importante recordar que, en el caso de tablas gestionadas por Spark, se eliminarán tanto los datos como los metadatos, mientras que, si la tabla no está gestionada por Spark, se eliminarán los metadatos (no se podrá volver a hacer referencia a la tabla eliminada), pero los datos originales no (por ejemplo, si la tabla se creó a partir de un fichero, este quedará intacto).

Vistas en Spark SQL

Un elemento auxiliar en Spark SQL son las vistas. Una vista especifica un conjunto de transformaciones sobre una tabla existente. Es decir, no son tablas, sino que definen el conjunto de operaciones que se harán sobre los datos almacenados en

cierta tabla para conseguir unos resultados. Las vistas se muestran como tablas, pero no guardan los datos en una nueva localización. Sencillamente, cuando se consultan, ejecutan las transformaciones definidas en ellas sobre la fuente de los datos. Por ejemplo, el siguiente ejemplo crea una vista:

```
CREATE VIEW flights_view AS
  SELECT * FROM flights
  WHERE dest_country_name = 'Spain'
```

La vista no contiene las filas de la tabla *flights* cuyo destino sea *Spain*, sino que únicamente almacena el plan de ejecución necesario para obtener esas filas de la tabla origen, de forma que las pueda mostrar cada vez que sea consultada. Si lo pensamos, una vista no es más que una transformación en Spark que nos devuelve un nuevo DataFrame a partir de otro DataFrame de origen; por tanto, no se ejecutará hasta que se haga una consulta que requiera obtener la vista. El principal beneficio es que evita escribir datos en disco repetidamente (como sería el caso de crear nuevas tablas). Existen diferentes tipos de vistas:

- **Vistas estándar,** que están disponibles de sesión en sesión, como la que veíamos en el ejemplo previo.
- **Vistas temporales**, que solo están disponibles en la sesión actual.

```
CREATE TEMP VIEW flights_view AS
  SELECT *
  FROM flights
  WHERE dest_country_name = 'Spain'
```

- **Vistas globales**, que son accesibles desde cualquier lugar de la aplicación Spark y no pertenecen a ninguna base de datos en concreto, pero se eliminan al final de la sesión.

```
CREATE GLOBAL TEMP VIEW flights_view AS
  SELECT *
  FROM flights
  WHERE dest_country_name = 'Spain'
```

Al crear una vista, podemos indicar que reemplace otra existente:

```
CREATE OR REPLACE TEMP VIEW flights_view AS
  SELECT *
  FROM flights
  WHERE dest_country_name = 'Spain'
```

Cuando ejecutamos la sentencia para crear una vista, vemos que no ocurre nada. Recordemos que dicha vista no se crea (no se ejecutan las transformaciones asociadas) hasta que se hacen consultas sobre ella, como, por ejemplo:

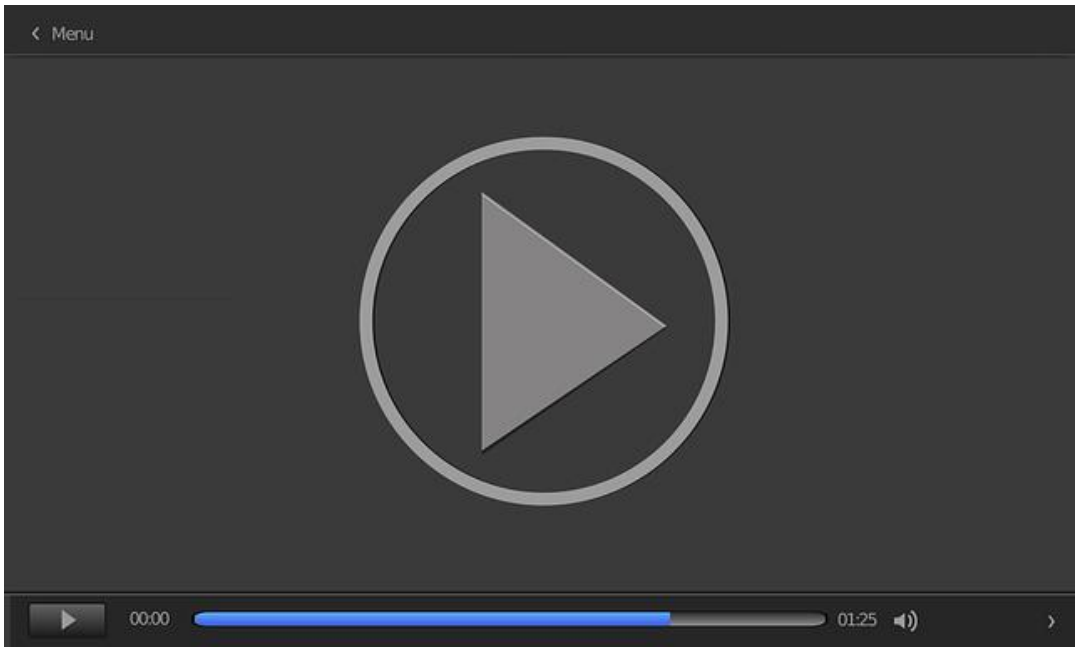
```
SELECT *
FROM flights_view; # ahora es cuando se ejecuta la vista.
```

Respecto a las consultas que se pueden hacer con las vistas, son las habituales de una tabla. Finalmente, podemos descartar una vista usando DROP:

```
DROP VIEW IF EXISTS flights_view;
```

Ideas clave

En el vídeo que exponemos a continuación, realizamos una introducción a Amazon Web Services (AWS). Concretamente, vamos a probar el servicio Elastic MapReduce con JupyterLab para ejecutar Spark.



Accede al vídeo:

https://unir.cloud.panopto.eu/Panopto/Pages/Embed.aspx?id=2ff50934-cd2e-4338-9285-ac8201622468

4.7. Ejemplo de Spark SQL

De nuevo, vamos a partir del ejemplo que hemos visto en el capítulo previo con la API básica y en este capítulo con la API estructurada. En esta ocasión, vamos a reproducirlo usando la API de Spark SQL.

```
# Supongamos que tenemos la SparkSession ya cargada.
# Empezamos por cargar los datos:
flightsDF = spark.read\
            .option("header", "true")\
            .csv("hdfs://<ip:port>/user/data/flights.csv")

# Para el uso de la API estructurada, es primordial registrar
# todos los DataFrames que queramos usar como tablas o vistas:
flightsDF.createOrReplaceTempView('flights')

# Calculamos los vuelos que llegan a cada aeropuerto
# de destino usando una sentencia SQL:
flights_dest_count = spark.sql('SELECT dest, COUNT(dest) AS dest_count FROM
flights GROUP BY dest ORDER BY dest_count DESC')


# De nuevo, para trabajar posteriormente con el resultado
# de esta consulta, necesitamos registrarla como vista:
flights_dest_count.createOrReplaceTempView('flights_dest_count')
flights_dest_count.show(5)
```

Obtenemos idéntico resultado que con la API estructurada. Y no solo eso, sino que cabe recordar que, a pesar de que el código es distinto porque usa API diferentes, al final ambas opciones se traducen en el mismo plan de ejecución, ya que Spark está concebido como un motor de procesamiento unificado.

```
+----+----------+
|dest|dest_count|
+----+----------+
| SFO|     12809|
| LAX|     10456|
| DEN|      9518|
| PHX|      8660|
```

```
| LAS|      8214|
+----+----------+
only showing top 5 rows
```

```
# Cargamos la información sobre los aeropuertos:
airportsDF = spark.read\
            .option("header", "true")\
            .csv("hdfs://192.168.240.4:9000/user/data/airport-codes.csv")
# Registramos el DataFrame como vista:
airportsDF.createOrReplaceTempView('airports')

# Llegados a este punto, solo resta hacer el join:
flights_dest_airports = spark.sql('SELECT a.name, f.dest_count FROM
flights_dest_count f JOIN airports a ON f.dest=a.iata_code ORDER BY
dest_count DESC')
flights_dest_airports.show(5)
```

Como podemos ver, obtenemos una vez más el mismo resultado. La diferencia recae en si el desarrollador tiene más manejo de sentencias SQL o de código .

```
+--------------------+----------+
|                name|dest_count|
+--------------------+----------+
|San Francisco Int...|     12809|
|Los Angeles Inter...|     10456|
|Denver Internatio...|      9518|
|Phoenix Sky Harbo...|      8660|
|McCarran Internat...|      8214|
+--------------------+----------+
only showing top 5 rows
```

Documentación oficial de Apache Spark

Apache Spark. Página web oficial: https://spark.apache.org/docs/latest/

Documentación *online*, detallada y de muy buena calidad sobre el sistema de computación Apache Spark.

Hadoop: the end of an era

Grishchenko, A. (2019, 23 de marzo). Hadoop: the end of an era [entrada de blog]. *Distributed Systems Architecture*. https://0x0fff.com/hadoop-the-end-of-an-era/

Hadoop es uno de los *frameworks* más importantes para el *big data*, cuyo propósito es almacenar grandes cantidades de datos y permitir consultas sobre estos, que se ofrecerán con un bajo tiempo de respuesta. Nació como iniciativa de Apache para dar soporte al paradigma de programación MapReduce, que fue inicialmente publicado por Google. En esta entrada de blog, se propone una interesante reflexión sobre su uso en la actualidad.

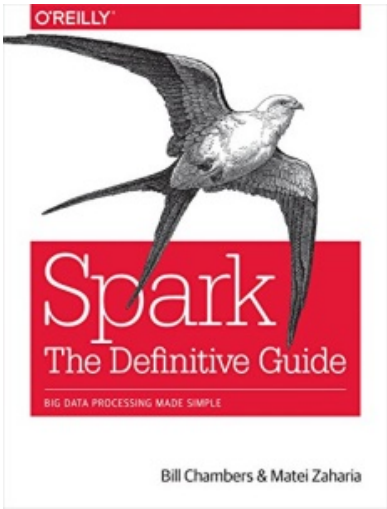
DATA + AI Summit

DATA + AI Summit. Página web oficial: https://databricks.com/sparkaisummit

Sitio web del congreso más famoso de Spark a nivel mundial, del que tienen lugar también versiones más reducidas en cada continente. Está organizado por Databricks, empresa que soporta el desarrollo de Spark.

Spark: the definitive guide

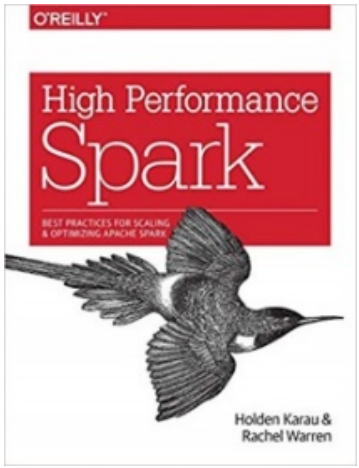
Chambers, B. y Zaharia, M. (2018). *Spark: the definitive guide*. O'Reilly.



Guía detallada de Spark, en su versión más actualizada. Contiene numerosos ejemplos y muestra exhaustivamente todas las capacidades de Spark.

High performance Spark

Karau, H. y Warren, R. (2017). *High performance Spark*. O'Reilly.



Manual avanzado sobre cómo escribir código optimizado en Spark.

Test

1. Elige la respuesta correcta respecto a los DataFrames de Spark:

   A. Un RDD es una envoltura de un DataFrame de objetos de tipo Row.

   B. Un DataFrame es una envoltura de un RDD de objetos de tipo Row.

   C. Un DataFrame es una envoltura de un objeto de tipo Row que contiene RDD.

   D. Ninguna de las respuestas anteriores es correcta.

2. Elige la respuesta correcta sobre los DataFrames de Spark:

   A. Puesto que representan una estructura de datos más compleja que un RDD, no es posible distribuirlos en memoria.

   B. Puesto que son un envoltorio de un RDD, suponen una estructura de datos que sigue estando distribuida en memoria.

   C. Son una estructura de datos no distribuida en memoria, al igual que los DataFrames de Python o los data.frames de R.

   D. Son una estructura de datos distribuida en disco.

3. ¿Qué mecanismo ofrece la API estructurada de DataFrames para leer datos?

   A. Método *read* de la Spark Session.

   B. Método *read* del Spark Context.

   C. No ofrece ningún método, sino que se utiliza la API de RDD para leer datos.

   D. Método *ingest* de la Spark Session.

4. ¿Es obligatorio especificar explícitamente el esquema del DataFrame cuando se leen datos de fichero?

A. No, porque solo se pueden leer ficheros estructurados como Parquet, que ya contienen información sobre su esquema.

B. Sí, porque, si no se indica el esquema, Spark no es capaz de leer ficheros CSV, ya que no sabe con qué tipo almacenar cada campo.

C. No, porque, si no se indica el esquema, Spark guardará todos los campos de los que no sepa su tipo como *strings*.

D. No, porque, si no se indica el esquema y se intenta leer ficheros sin esquema implícito, Spark lanzará un error.

5. Seleccione la respuesta incorrecta: ¿Por qué es aconsejable utilizar DataFrames en Spark en lugar de RDD?

A. Porque son más intuitivos y fáciles de manejar a alto nivel.

B. Porque son más rápidos, debido a optimizaciones realizadas por Catalyst.

C. Porque los DataFrames ocupan menos en disco.

D. Las respuestas A y B son correctas.

6. Tras ejecutar la operación `b = df.withColumn("nueva", 2*col("calif"))`:

A. El DataFrame contenido en `df` tendrá una nueva columna, llamada nueva.

B. Llevaremos al *driver* el resultado de multiplicar 2 por la columna `calif`.

C. El DataFrame contenido en b tendrá una columna más que `df`.

D. El DataFrame contenido en b tendrá una única columna llamada nueva.

7. ¿Cuál es la operación con la que nos quedamos con el subconjunto de filas de un DataFrame que cumplen una determinada condición?

A. sample.

B. filter.

C. map.

D. show.

8. Las API estructuradas de DataFrames y Spark SQL…

A. Son API que no se pueden combinar: una vez que se empieza a usar una de ellas, se tienen que hacer todas las tareas con la misma API.

B. Se pueden aplicar funciones de la API de DataFrames sobre el resultado de consultas de Spark SQL.

C. Se pueden aplicar el método `sql` para lanzar consultas SQL sobre DataFrames sin registrar.

D. Ninguna de las opciones anteriores es correcta.

9. La transformación `map` de Spark…

A. No se puede aplicar a un DataFrame porque pertenece a la API de RDD.

B. Se puede aplicar a un DataFrame porque pertenece a la API estructurada de DataFrames.

C. Se puede aplicar a un DataFrame porque envuelve un RDD al que se puede acceder mediante el atributo `rdd`.

D. No existe en Spark; `map` es una acción.

10. Para utilizar Spark SQL, es necesario…

A. Utilizar la función `sql` del objeto `SparkContext`.

B. Utilizar la función `sql` del objeto `SparkSession`, a fin de ejecutar la consulta SQL sobre el DataFrame directamente.

C. Registrar el DataFrame sobre el que se quieran ejecutar las consultas SQL como tabla o vista, antes de ejecutar cualquier consulta.

D. Ninguna de las respuestas anteriores es correcta.