Ingeniería para el Procesado Masivo de Datos

Dr. Pablo J. Villacorta

Tema 4. Apache Spark II

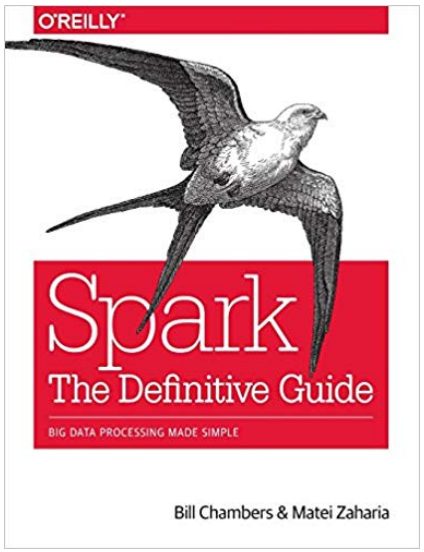
Noviembre de 2022

unir LA UNIVERSIDAD EN INTERNET

Universidad Internacional de La Rioja

Objetivos del tema

- Conocer la API estructurada 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.

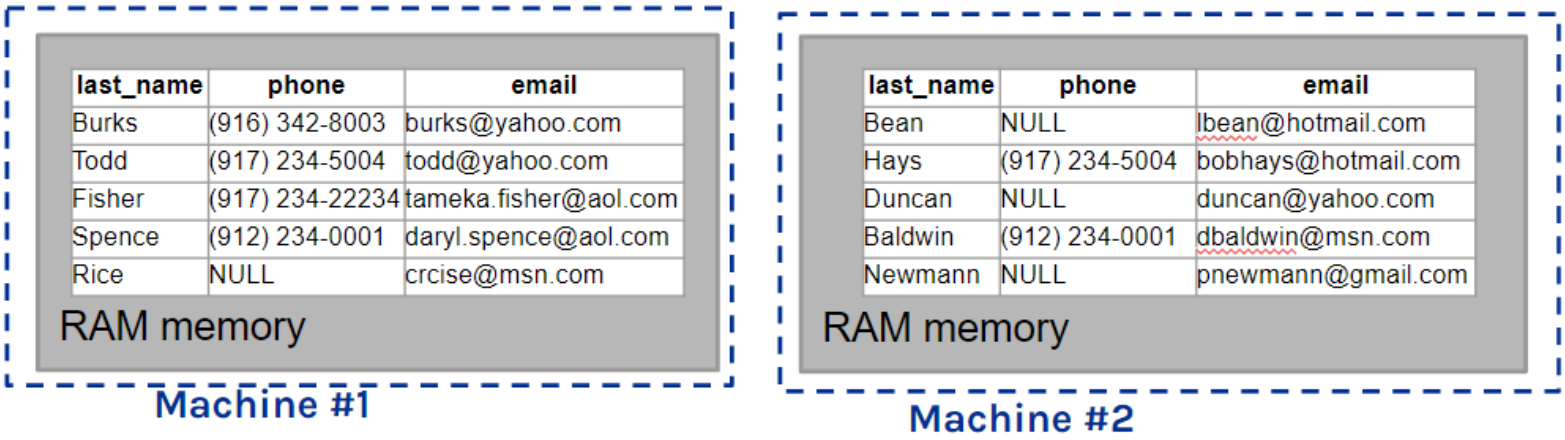

O'REILLY
Spark
The Definitive Guide
BIG DATA PROCESSING MADE SIMPLE
Bill Chambers & Matei Zaharia

DataFrames y Spark SQL

- El cálculo con RDD es tedioso y de bajo nivel. **Spark SQL** nace para manejar datos intrínsecamente *estructurados*, como si manejásemos tablas distribuidas en memoria
  - Es posible usar Spark SQL para procesar JSON, pero sólo *tras asignarle un esquema* para convertirlo a tabla
- *DataFrame*: RDD + esquema (tipos y nombres de columnas) que se maneja *como si fuese* una tabla de una base de datos.
  - Esquema explícito, o bien inferido por Spark
  - Un DataFrame no es más que un RDD de objetos de tipo *Row* («fila»), el cual es similar a un array: cada fila se asemeja a un vector de elementos con nombre
  - Podemos obviar este detalle ya que siempre vamos a **manejarlos utilizando la API Estructurada** (operaciones propias de los DataFrames)

- Introducidos en Spark 1.6 (año 2016). A partir de Spark 2.0 se considera la estructura de datos fundamental para la API de Spark

DataFrames: "tablas distribuidas en memoria"



DataFrame

	Name	Age	City	Country
Part #1	Pablo	24	Barcelona	Spain
	Emma	32	Alabama	USA
Part #2	Tony	28	Frankfurt	Germany
	Jose	35	Madrid	Spain

=

RDD de Rows

Part #1	Row(Name=Pablo, Age=24, City=Barcelona, Country=Spain) Row(Name=Emma, Age=32, City=Alabama, Country=USA])
Part #2	Row(Name=Tony, Age=28, City=Frankfurt, Country=Germany) Row(Name=Jose, Age=35, City=Madrid, Country=Spain)

```
resultadoDF = personas.select("Age", "Country").where("Age >= 18 and Country == 'Spain' ")
```

Manipulación de DataFrames: Spark SQL API

- Existe una API para manejar DataFrames: ***API estructurada*** (paquete **SparkSQL)**
- Vamos a ver las transformaciones y acciones más frecuentes sobre DataFrames

Acción *count():* cuenta las filas del DF y devuelve al driver un entero:

Acción *show(n=20):* lleva al driver el número indicado de filas (20 si no se indica nada) y lo imprime por pantalla. **No devuelve ningún resultado (devuelve *None* de Python)**

Acción *collect():* lleva al driver **todas** las filas del DF como una **lista de objetos *Row*, y la devuelve como resultado. PRECAUCIÓN:** deben caber en el driver o dará error!

Acción *take(n):* lleva una lista de *n Rows* al driver. Los *n* primeros si el DF fue ordenado

```
numFlights = flightsDF.count()    # materializa flightsDF
flightsDF.cache()                 # para la próxima, no liberar
flightsDF.show(30)                # materializa flightsDF parcialmente
listaTodos = flightsDF.collect()  # una parte ya estaba materializada
flightsOrdenados = flightsDF.sort("ArrDelay") # Aquí no hace nada
menorRetraso = flightsOrdenados.take(10) # Aquí se hace sort y take
for r in menorRetraso:  # recorremos una lista de Python
    print(r.ArrDelay)   # acceso a columnas de un Row con '.'
```

Manipulación de DataFrames: Spark SQL API

Transformación *select*

- Recibe los nombres de las columnas que queremos seleccionar y devuelve otro DF que solo contiene esas columnas Todas las columnas deben existir!

```
transformedDF = flightsDF.select("Year", "Month",
                        "DayofMonth", "ArrTime", "FlightNum")
```

- También podemos usarlo para crear nuevas columnas al vuelo:

```
from pyspark.sql import functions as F

resultDF = flightsDF.select(
  F.col("Year"),
  F.col("Distance"),
  (1.6*F.col("Distance")).alias("DistKm")
)

resultDF.show()
```

Year	Distance	DistKm
2008	2400	3840
2008	3200	5120
2008	1500	2400
2008	4500	7200

Manipulación de DataFrames: Spark SQL API

Transformación *where / filter*

- Devuelve un nuevo DF que contiene solo las filas que cumplen cierta condición booleana sobre una o más columnas
  - filter() y where() son exactamente equivalentes
  - El argumento puede ser un string con una condición en SQL puro
  - También puede ser una condición booleana utilizando la API de SparkSQL API sobre las columnas

```
from pyspark.sql import functions as F
transformedDF = flightsDF\
    .filter("DayOfMonth < 20 or Origin = 'LAX'")\ # SQL puro
    .filter("Carrier is not null")\              # SQL puro
    .filter(~(F.col("FlightNum").isNull()))\ # API estructurada
    .filter(1.6*F.col("Distance") > 5000)\  # API estructurada
    .filter(F.col("Delay") > F.col("DepDelay") + F.col("ArrDelay"))\
    .where("Delay > 15")          # SQL puro
```

Manipulación de DataFrames: Spark SQL API

Transformación *withColumn*

- Crea una nueva columna de otras existentes. Devuelve un nuevo DF que contiene todas las columnas existentes más la nueva por la derecha

```
from pyspark.sql import functions as F
transformedDF = flightsDF.withColumn("DistKm", 1.6*F.col("Distance"))
```

- Si el nombre corresponde a una columna ya existente, estaremos reemplazando (creándola de nuevo) en el nuevo DF en su misma posición

- Para la nueva columna se puede utilizar cualquier **operación entre columnas ya** implementada en Spark SQL en el paquete **`pyspark.sql.functions`** (todas son operaciones ya distribuidas)
  - Se suelen encadenar múltiples operaciones withColumn una tras otra.
  - Las nuevas columnas se van creando por la derecha en el mismo orden en el que vamos encadenando las operaciones.

Manipulación de DataFrames: Spark SQL API

Transformación *withColumn*

```
from pyspark.sql import functions as F

transformedDF = flightsDF\                    # primero creamos una nueva
  .withColumn("DistKm", 1.6*F.col("Distance"))\      # columna
  .withColumn("TotalDelay", F.col("DepDelay")+ F.col("ArrDelay"))\
  .withColumn("Year", F.lit(2009)) # reemplazar columna existente
```

Year	…	…	DistKm	TotalDelay
2009	…	…	3840	26
2009	…	…	5120	17
2009	…	…	2400	32
2009	…	…	7200	7

Manipulación de DataFrames: Spark SQL API

Transformación *drop*

- Elimina una o varias columnas, por nombre. Si no existen, NO da error.
- Devuelve un nuevo DF sin esas columnas (el DF original no se modifica)

Transformación *withColumnRenamed*

- Devuelve un nuevo DF donde una columna se ha renombrado en su misma posición

```
transformedDF = flightsDF\
 .withColumnRenamed("Delay", "MinutesLate")\ # nuevo nombre
 .drop("Carrier", "FlightNum", "inventedName") # es correcto
```

Transformación *sort / orderBy*

- Devuelve un nuevo DF ordenado en base a las columnas indicadas. Se utiliza la primera, y las demás solo para desempatar en caso de empate

```
flightsDF.sort("Origin", "flightDate")\
         .show(10) # los 10 primeros
```

Manipulación de DataFrames: Spark SQL API

Algunas funciones para operar con columnas

- Función ***col*** : construye un objeto Column de un nombre de columna
- Función ***alias***: devuelve un nuevo objeto Column renombrado
- La más frecuente es ***when***, que devuelve un objeto columna o una constante en función de condiciones booleanas sobre otras columnas
  - Se indica el valor que debe tener cada fila, y puede ser una constante o también el resultado de operar con columnas
  - Muy habitual para re-categorizar variables existentes o para calcular variables nuevas (ingeniería de variables, *feature engineering*)

```
from pyspark.sql import functions as F
transformedDF = flightsDF\
 .withColumn("retraso",
      F.when(F.col("ArrDelay") "puntual")\
       .when((F.col("ArrDelay") > 15) & (F.col("ArrDelay") < 60),
             "impuntual")\
       .otherwise("inaceptable"))\
 .withColumn("ArrDelay", F.when(F.col("ArrDelay") < 0, 0)\
                          .otherwise(F.col("ArrDelay")))
```

Manipulación de DataFrames: Spark SQL API

Algunas funciones para operar con columnas

- Agregación: F.mean("colName"), F.min, F.max, F.stddev, F.corr , …
- Columna constante: F.lit("valor")

Transformación para eliminar filas duplicadas y valores nulos

```
resultDF = df1.distinct()   # duplicado = coincidencia en todas las columnas
resultDF = flightsDF.dropDuplicates(subset = ["Origin", "Dest"])
resultDF = flightsDF.dropna()  # quitar si hay NA/NULL en alguna columna
resultDF = flightsDF.dropna(subset = ["ArrDelay"]) # sólo si hay NA en ArrDelay
```

Transformación *union, intersect, except* (todos los DF mismo esquema)

```
resultDF = df1.union(df2).intersect(df3).except(df4)
```

Transformación *groupBy* seguida de *agg*

- Transformación para agregar por grupos

```
agregadosDF = flightsDF.groupBy("Origin", "Dest").agg(
              F.mean("ArrDelay").alias("retrasoMedioRuta"),
              F.count("*").alias("recuento"))
```

Manipulación de DataFrames: Spark SQL API

- Además de la API, existe un método para ejecutar tal cual consultas en SQL: el método **sql**("consultaComoString") que se invoca sobre la SparkSession

```
# Creamos una vista volátil que desaparecerá al cerrar Spark
flightsDF.createOrReplaceTempView("flights") # metadatos en Hive
resultDF = spark.sql("select Origin from flights
                      where ArrDelay > 15")
```

- Soporta consultas SQL complejas y las traduce a trabajos de Spark optimizados
  - Recordemos: motor de ejecución único
  - La API estructurada es igual de eficiente que efectuar consultas en SQL a Spark porque al final se traducen a un mismo plan de ejecución abstracto.

Lectura y escritura de DataFrames

- Lectura de datos de un datasource, que por defecto es HDFS

  `myDF = spark.read.format(<formato>).load(“/path/to/hdfs/file”)`

  <formato> = “parquet” | “json” | “csv” | “orc” | “avro”, y *spark* es el objeto SparkSession

- Suele estar disponible un atajo con **`spark.read.<format>("/path/to/file")`** aunque no todos los formatos lo tienen

```
df1 = spark.read.option("inferSchema", "true")\
                .csv("/path/to/file") # infiere

df2 = spark.read.option("inferSchema", "false")\
                .csv("/path/to/file") # todas string

from pyspark.sql import types as T
myschema = T.StructType([
    T.StructField("columna1", T.DoubleType(), nullable = False),
    T.StructField("columna2", T.DateType(), nullable = False)
])
df3 = spark.read.option("header", “true”)\
           .option("delimiter", "|")\
           .schema(myschema).csv("/path/to/file")
```

User Defined Functions (UDF)

- Función que recibe datos de cada fila y devuelve un resultado. Se aplica a cada fila
- Similar a un *map* pero …
  - Aplicado a un DataFrame, donde intervienen solo ciertas columnas
  - Recibe el tipo simple de cada columna implicada (en vez de un objeto *Row*)
  - Se utiliza dentro de **withColumn()** para *crear una nueva columna a un DF existente, con el resultado de aplicar a cada fila nuestra función, en la que intervienen solo ciertas columnas de la entrada*.
  - En cambio, *map* devuelve un DF completamente nuevo (no añade columnas)

```
def computeInitialGroup(year1, year2): # year1 y year2 son strings
    if (year1 != "None_or_-1" and year1 != "Other"):
        return year1
    elif (year2 != "None_or_-1" and year2 != "Other"):
        return year2
    else:
        return "Unknown"

udfinitial_group = F.udf(computeInitialGroup, StringType())
# ... pero invocamos a nuestra UDF pasándole 2 columnas (de tipo string)
myDF = spark.read.parquet("/tmp/misdatos.parquet")
newDF = myDF.withColumn("initialGroup", udfinitial_group(
            F.col("year1Col"), F.col("year2Col"))
```

…eso es todo por hoy…

:-)

Cualquier duda, consulta o comentario:

mensaje a través de la plataforma!

www.unir.net