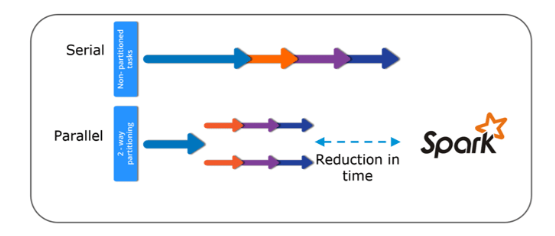
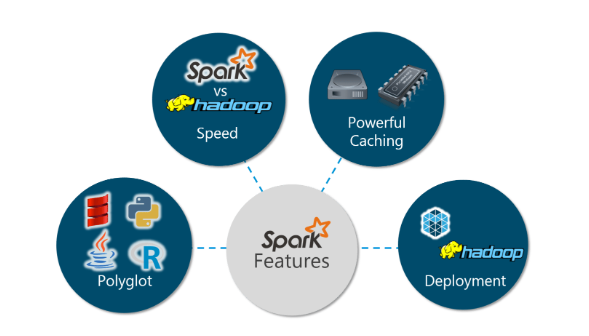
Apuntes SPARK SQL

**Introducción a Spark**

Apache Spark es un marco informático de clúster de código abierto para el procesamiento en **tiempo real**. Tiene una próspera comunidad de código abierto y es el proyecto Apache más activo en este momento. Spark proporciona una interfaz para programar clústeres completos con paralelismo de datos implícito y tolerancia a fallos.

Fue construido sobre Hadoop MapReduce y extiende el modelo MapReduce para usarlo de manera más eficiente y con más tipos de cálculos.

Las características de Spark son las siguientes:



**Políglota:**

Spark proporciona un API de alto nivel en Java, Scala, Python y R. El código de Spark se puede escribir en cualquiera de estos cuatro lenguajes.

Proporciona un shell en Scala y Python. Se puede acceder al Shell desde el directorio instalado:

* **Scala** a través de ./bin/spark-shell
* **Python** a través de ./bin/pyspark.

**Velocidad:**

Spark se ejecuta hasta **100 veces más rápido** que Hadoop MapReduce para el procesamiento de datos a gran escala. Spark puede lograr esta velocidad a través de la partición controlada. **Administra los datos mediante particiones** que ayudan a paralelizar el procesamiento de datos distribuidos con un tráfico de red mínimo.

**Evaluación perezosa:**

Apache Spark retrasa su evaluación hasta que sea absolutamente necesario. Este es uno de los factores clave que contribuyen a su velocidad.

Para las transformaciones, Spark las agrega a un DAG (Gráfico a cíclico dirigido) de cálculo y solo cuando el controlador solicita algunos datos, este DAG realmente se ejecuta.

**Aprendizaje automático:**

**MLlib de Spark** es el componente de aprendizaje automático, útil cuando se trata de procesamiento de Big Data. Erradica la necesidad de usar múltiples herramientas; una para procesamiento y otra para aprendizaje automático. Spark proporciona a los ingenieros y científicos de datos un motor potente y unificado que es rápido y fácil de usar.

Spark SQL

Antes de continuar, iniciemos Apache Spark en nuestros sistemas y acostumbrémonos a los conceptos principales de Spark, como **Spark Session**, **Data Sources**, **RDD**, **DataFrames** y otras bibliotecas.

**Spark Session:**

En versiones anteriores de Spark, **SparkContext** era el punto de entrada para Spark. Para todas las demás API, necesitábamos usar diferentes contextos.

Para la transmisión, necesitábamos **StreamingContext**, para SQL **sqlContext** y para Hive **HiveContext**.

Para resolver este problema, **SparkSession** entró en escena. Es esencialmente una combinación de SQLContext, HiveContext y el futuro StreamingContext.

**DataFrame**

Un DataFrame es un conjunto de datos organizado en columnas con nombre. Es conceptualmente **equivalente a una tabla** en una base de datos relacional o un **marco de datos en R/Python**, pero con optimizaciones.

Los DataFrames se pueden construir a partir de una amplia gama de fuentes, como: archivos de datos estructurados, tablas en Hive, bases de datos o RDD existentes.

Spark SQL - RDD

El conjunto de **datos distribuido resilente** (RDD) es una estructura de datos fundamental de Spark. Es una **colección distribuida inmutable** de objetos. Cada conjunto de datos en RDD se divide en particiones lógicas, que se pueden calcular en diferentes nodos del clúster.

Los RDD pueden contener cualquier tipo de objetos de Python, Java o Scala, incluidas clases definidas por el usuario.

Las **características** principales de los RDDs son:

* Abstracción de datos.
* Los RDDs están **particionados en los nodos** del cluster.
* Se suelen crear **a partir de un fichero del HDFS**.
* Usan la **evaluación** **perezosa**.
  + - * Los RDDs usan evaluación perezosa en sus **transformaciones**.
      * Mantiene todas las transformaciones en un DAG.
      * Cuando se lanza **una acción, se resuelve el grafo**.

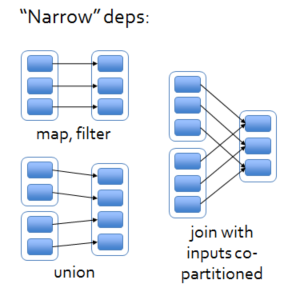
Para empezar a programar debemos conocer SparkConf y SparkContext:

* El objeto **SparkContext** especifica cómo vamos a acceder a nuestro cluster.
* El objeto **SparkConf** contiene la información sobre nuestra aplicación.

**Las Transformaciones y acciones**

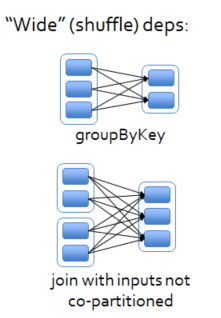
Las diferencias entre acciones y transformaciones son:

* **Transformaciones**
  + Crea un nuevo RDD a partir de otro existente.
  + Map es una transformación
* **Acciones**
  + Genera un valor que es mandado al Driver.
  + Reduce es una acción

Existen **dos tipos** de transformaciones:

Narrow transformation: todos los datos para calcular los registros de **una partición** se encuentran en una única partición del RDD.

P.e. map (), filter(), flatMap(), Union()

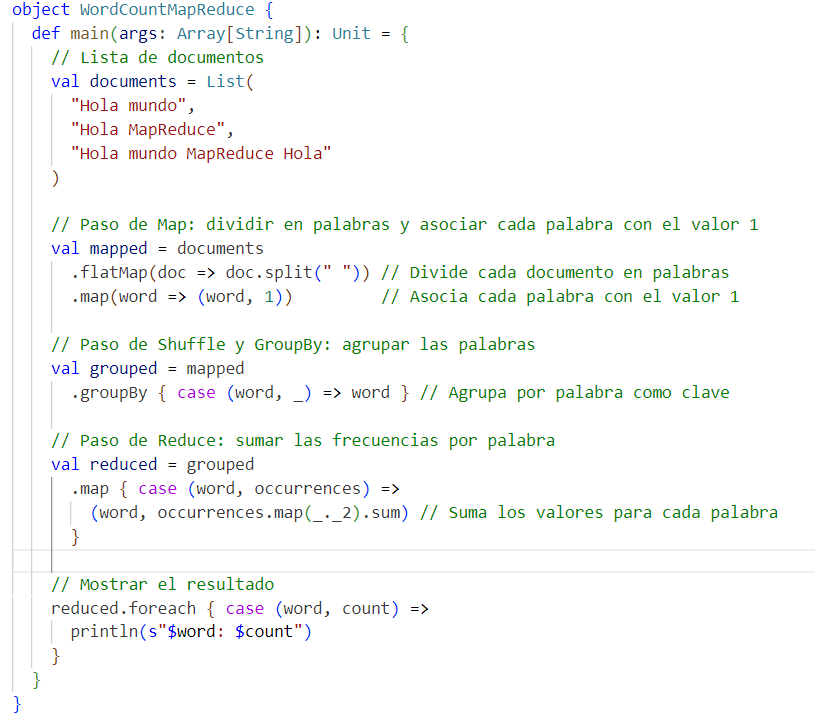


**Wide transformation:** todos los datos para calcular los registros en una partición pueden residir **en muchas particiones del RDD** principal.

P.e. groupbyKey(), reducebyKey(), intersection(), distinct(), sortByKey(), join(), cogroup(), coalesce()

MAP REDUCE

**Cuenta palabras**

****

**Explicación paso a paso**

* **Entrada**: La lista documents contiene los textos de entrada. Esto podría ser un fichero del que se quiera contar las palabras.

**Paso de Map:** Se divide cada el documento en palabras.

* flatMap: divide cada documento en palabras
* Map: asocia cada palabra con el valor 1 (palabra, 1).

List(("Hola", 1), ("mundo", 1), ("Hola", 1), ("MapReduce", 1), ("Hola", 1), ("mundo", 1), ("MapReduce", 1), ("Hola", 1))

**Shuffle y Agrupación:** Se agrupan las palabras iguales

* groupBy: para agrupar las palabras similares. Esto genera un mapa donde la clave es la palabra y el valor es una lista de las ocurrencias.

Map(

"Hola" -> List((Hola,1), (Hola,1), (Hola,1), (Hola,1)),

"mundo" -> List((mundo,1), (mundo,1)),

"MapReduce" -> List((MapReduce,1), (MapReduce,1))

)

**Paso de Reduce:** Recorre el mapa agrupado y suma los valores de cada lista.

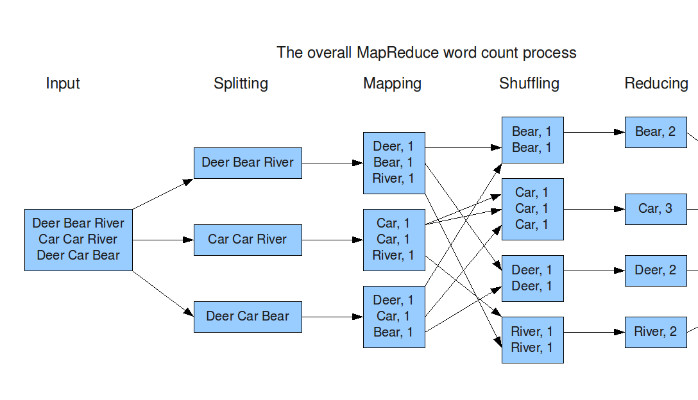
Map("Hola" -> 4, "mundo" -> 2, "MapReduce" -> 2)

**Resultado:**

Hola: 4

mundo: 2

MapReduce: 2



EJEMPLOS

**//Importamos las clases necesarias**

import org.apache.spark.SparkConf

import org.apache.spark.SparkContext

import org.apache.spark.rdd.RDD

en este punto nos ha creado un sparkSession que se llama **spark** y sparkContext que se llama **sc**.

**//Para ver la configuración por defecto del sc**

sc.getConf.getAll()

**//Crear un RDD a partir de un array de strings**

val array= Array(“me”,”gusta”,”el”, “BigData”)

val arrayRDD = sc.parallelize(array)

**//Para visualizar nuestro contenido de RDD**

arrayRDD.collect()

**//Crear un RDD a partir de un fichero de texto**

val file = sc.textFile(“/home/misPracticas/spark”)

file.collect()

**Transformaciones**

**//Transformacion: filter(). Se usa para hacer filtros de los RDDs**

val filtrado = file.filter(line => line.contains(“curso”))

filtrado.collect()

**//Transformación: map() Se usa cuando quieres hacer una //transformación a todo tu RDD.**

**//p.e: Pasar todas la palabras del RDD a mayusculas**

val cadenaMayusculas = arrayRDD.map(palabra => palabra.toUpperCase())

cadenaMayuscula.collect()

**//Transformación: flatMap() es como el map() pero la salida es un //objeto compuesto**

**//p.e una lista. Saco por un lado la palabra en mayuscula y por el otro su //tamaño**

val mayusLen = arrayRDD.flatMap(p=> List(p.toUpperCase(), p.length))

mayusLen.collect()

**//Transformación: union() hace la union de dos RDDs**

val miUnion= arrayRDD.union(cadenaMayusculas)

miUnion.collect()

**//Transformación: distinct() en un RDD devuelve lo distintos valores**

val miDistinct = miUnion.distinct()

miDistinct.collect()

**//Transformaciones: reduceByKey y groupByKey. Para agrupar por un //campo.**

**//ejemplo de cuenta palabras: primero hago el map para dejarlo listo //para agrupar y reducir**

val miMap= arrayRDD.map (p=> (p,1))

miMap.collect()

val miGrupo = miMap.groupByKey()

miGrupo.collect()

val miReduce = miGrupo.reduceByKey(\_+\_)

ó

val miReduce = miGrupo.reduceByKey ((a,b) => a+b)

miReduce.collect()

**//Transformación: shortByKey para ordenar ascendente (true) o //descendente (false)**

val miOrden = miGrupo.shortByKey(false)

miOrden.collect()

**//Definición de funciones**

def tamanyo( s : String) : (String , Int) ={

return (s, s.length)

}

**//Ejemplo de llamada a la función**

val miMap = arrayRDD.map (tamanyo(\_))

miMap.collect()

**Acciones**

**//Acción: collect es para mostrar. (como el print)**

miRdd.collect()

**//Acción: first() nos devuelve el primer elemento**

miRdd.first()

**//Acción:count con devuelve el número de elementos**

miRDD.count

val miSuma = miRDD.count + 5

**//Acción: reduce() hace una operación a todos los elementos del RDD**

val miReduce= miArrayNumRDD.reduce (\_+\_)

val miReduce= miArrayNumRDD.reduce (\_\*\_)

miReduce.collect ()

**//Acción:take() nos devuelve n elementos.**

val misCuatroElementos = miArray.take(4)

misCuatroElementos.collect()

**//Accion: max y min devuelve el máximo y el mínimo**

val miMaximo = miArray.max

val miMinimo = miArray.min

**//Accion: countByKey cuenta por key. Es para hace un “cuenta //palabras”**

val miMap = miRDD.map(p=> (p,1))

val miCount = miMap.counByKey()

miCount.collect()

**//Acción: foreach**

miRdd.foreach( p=> println(“La palabra del RDD es ” + p))

Persistencia y acumuladores

**Persistencia:**

Spark nos permite elegir donde tener los datos con los que vamos a trabajar; en disco y/o en memoria. Tenerlos en memoria nos proporciona más velocidad, pero en ambas dos formas de trabajar es tolerante a fallos.

Los tipos de persistencia son los siguientes:

✔ MEMORY\_ONLY

✔ MEMORY\_AND DISK

✔ MEMORY\_AND\_DISK\_SER

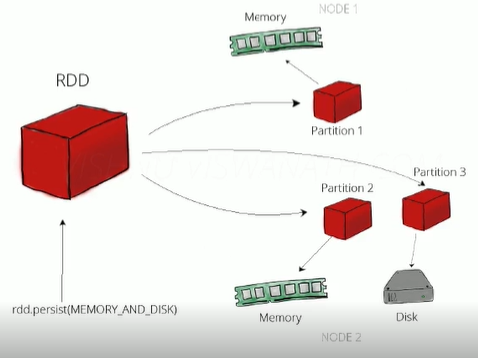
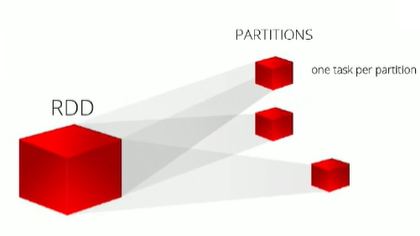
✔ (ser significa serializable)

✔ DISK\_ONLY

✔ MEMORY\_ONLY\_SER

✔ MEMORY\_ONLY\_SER

✔ MEMORY\_ONLY2



**Acumuladores:**

Los acumuladores son variables globales que toda parte del código, aun estando en distintos clusters, lo ven y pueden acceder a él.

Estos acumuladores pueden ser de tipo Long o tipo Double.

**Ejemplos:**

Como vamos a seguir trabajando con RDD todo lo anterior nos vale.

Vamos a cargar un fichero en un RDD, a continuación, vamos a persistir el RDD en disco y luego en memoria.

**//Importamos las clases**

import org.apache.spark.SparkConf  
import org.apache.spark.SparkContext  
import org.apache.spark.rdd.RDD

import org.apache.spark.storage.StorageLevel

**//trabajamos con el fichero persistenciaFile**

**//Leer el fichero**

val fileMemory = sc.textFile (“/home/persistenciaFile”)

**//Comprobamos tiempos con persistencia en MEMORIA**

fileMemory.persist(StorageLevel.MEMORY\_ONLY)

val timestampInicio: Long = System.currentTimeMillis

val mayor = fileMemory.flatMap(linea => linea.split(“ ”)).filter(p=>p.length > 6)

mayor.count()

val timestampFin: Long = System.currentTimeMillis

println (“Tiempo: ” + (timestampFin - timestampInicio))

**//Quitamos la persistencia**

fileMemory.unpersist()

**//Comprobamos tiempos con persistencia en DISCO**

fileMemory.persist(StorageLevel. DISC\_ONLY)

val timestampInicio: Long = System.currentTimeMillis

val mayor6 = fileMemory.flatMap(linea => linea.split(“ “)).filter(p=>p.length > 6)

mayor6.count()

val timestampFin: Long = System.currentTimeMillis

println (“Tiempo: ” + (timestampFin - timestampInicio))

**//Quitamos la persistencia**

fileMemory.unpersist()

**//Acumulador a partir del sparkContext (sc)**

val miAcumulador = sc.longAccumulator (“NumPalabras”)

mayor6.foreach (p=> miAcumulador.add(1))

miAcumulador.value

2.4. Dataframe y Dataset

Dataframes y datasets

**Dataset (objetos):** Es una colección de datos distribuidos que tienen una estructurados. Los dataset se introdujeron a partir de Spark 1.6. Actualmente la API está disponible para Java y Scala (en python no existe).

A diferencia de los RDD, estos tienen una estructura. Es decir, puedo tener un fichero con los datos de empleado; nombre, puesto, edad. Se puede crear un objeto empleado y crear un dataset con la estructura del objeto

**Dataframe (columnas):** Son datasetorganizados en columnas.

Tanto para usar los dataset como los dataframes se necesita spartSession. Pueden convivir juntos, es decir, en el mismo código puede tener dataframes y datasets.

**//Crear una sparkSession**

val spark = SpartSession.buider()

**Ejemplo dataframe:**

**//importamos las clases necesarias**

import org.apache.spark.sql.SparkSession

**//creamos y configuramos**

val spark = SpartSession.buider().appName(“miApp”).master(“local”).getOrCreate()

**//leemos un fichero en json y lo convertimos en daframe**

var miDF = spark.read.json (“/home/strangerCharacters.json”)

**//para ver el contenido del DF**

miDF.show()

**//Para ver la estructura del DF**

miDF.columns

**//Para ver el esquema de DF**

miDF.printSchema

**//Para seleccionar unas columnas**

val miNombresDF = miDF.select (“nombre”)

miNombresDF.show()

**//Para seleccionar dos columnas...**

val miNombreEdad = miDF.select (“Nombre”,”Edad”)

miNombreEdad.show()

**//Para exportarlo en un fichero Parquet (no legible por nosotros)**

miNombreEdad.write.save(“/home/miParket”)

**//Para exportarlo en un fichero Json**

miNombreEdad.write.json(“/home/miJSON”)

**//Filtros**

val miDFJovenes= miDF.filter ($”edad” < 15)

miDFJovenes.show()

**//Para sacar el primer elemento**

miDF.first()

**//Para sacar un número en concreto de filas (como el take de RDD)**

miDF.head(5)

**//Para contar elementos**

miDF.count()

**//Para agrupar por edad. Nos dice cuantos personajes hay por cada edad**

miDF.groupBy(“edad”).count().show()

**//Información básica del DF**

miDF.describe().show()

**Ejemplo dataset:**

**Para este ejemplo:**

* **poner el fichero en forma de txt**
* **cargar en un RDD**
* **transformar el RDD en un dataset.**

**//Importamos las clases**

import org.apache.spark.SparkConf  
import org.apache.spark.SparkContext  
import org.apache.spark.rdd.RDD

import org.apache.spark.storage.StorageLevel

import org.apache.spark.sql.SparkSession

import org.apache.spark.sql.Dataset

import org.apache.spark.sql.DataFrame

import spark.implicits.\_

import org.apache.spark.sql.Row

**//creamos y configuramos**

val spark = SpartSession.buider().appName(“miApp”).master(“local”).getOrCreate()

val sc = spark.sparkContext

**//Cargamos el fichero al RDD**

val miRDD = sc.textFile (“/home/strangersCharacters.txt”)

miRDD.collect()

**//Le damos la estructura: Primero creamos la clase**

case class Personaje (nombre: String, edad: Long, sexo: String)

**//Exportamos el fichero a la clase.**

**//Por cada linea creo un elemento tantos elementos como elemento separados**

**//por coma haya**

val partes = miRDD.map(\_.split(“,”))

partes.collect()

**//Por cada array creo un Personaje**

val personajes = partes.map(arr => Personaje(atr(0), atr(1).trim.toInt, atr(2))).toDS()

personajes.show()

**//Consultamos los personajes mayores de 13 años**

personajes.where($”edad” > 13).show()

**//Para la persistencia**

import org.apache.spark.storage.StorageLevel

personajes.persist(StorageLevel.DISK)

**//Para pasar el DS a RDD**

personajes.rdd

**//Pasar de DS a DF**

val personaDF = personajes.toDF(“nombre”,”edad”,”sexo”)

**//Para devolver una lista con el numero de elementos que queramos**

personajes.takeAsList(5)

Vistas

Las vistas permiten lanzar consultas SQL sobre DataFrames. Existen dos tipos de

vistas. Por un lado, tenemos las vistas temporales y por otro las vistas globales:

**Vistas temporales:**

Este tipo de vistas solo existe en la sesión en la que la creamos

DF.createOrReplaceTempView(“nombre\_vista”)

**Vista global:**

Este tipo de vista están disponibles para cada sesión de spark que creemos.

DF. creteOrReplaceGlobalTempView(“nombre\_vista”)

spark.sql(“SELECT \* FROM **global\_temp**.nombre\_vista”)

**Ejemplos con Vistas**

Con el fichero de tweeter que tenemos

tweetsJson.ceateOrReplaceTempView(“tweets”)

**//creamos DS con una sentencia SQL sobre la vista**

val usuarios = spark.sql (“select user.screen\_name from tweets”)

**//Cuidado con los group by: NUEVA SINTAXIS**

val miDF = spark.sql (“select first(campo1), first(campo2), sum(campo3) group by campo2”)

Estructuras de datos

Spark SQL permite trabajar con una amplia variedad de tipos de datos y estructuras. Estos incluyen tipos primitivos, complejos y personalizados que facilitan el procesamiento de datos estructurados y semiestructurados.

**Tipos de Datos Primitivos**

Tipo de datos básicos que se utilizan en DataFrames y SQL:

* **IntegerType**: Enteros.
* **StringType**: Cadenas de texto.
* **DoubleType**: Números decimales.
* **BooleanType**: Valores booleanos.
* **DateType y TimestampType**: Fechas y timestamps.

**Tipos de Datos Complejos**

Spark SQL soporta estructuras avanzadas como:

* **StructType:** Permite anidar estructuras (como JSON).
* **ArrayType:** Permite manejar listas de valores dentro de una columna.
* **MapType:** Permite manejar pares clave-valor dentro de una columna.

**Tipos de Datos Avanzados**

* **BinaryType:** Se usa para manejar datos binarios como imágenes o archivos en formato binario.
* **TimestampType y DateType:** Trabaja con fechas y timestamps para análisis temporal.

**Dataset API con Tipos Personalizados**

**Vistas o Interoperabilidad con SQL**

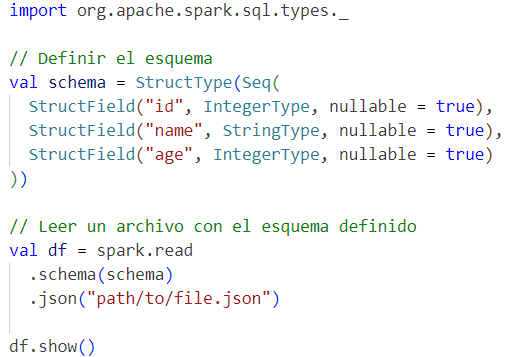
Structs

Los Structs se utiliza principalmente para definir la estructura de los datos, especialmente cuando trabajamos con DataFrames y Datasets.

**Structs** son fundamentales en Spark porque permiten describir y manejar datos con esquemas definidos, facilitando las operaciones como la consulta, transformación, y validación de datos.

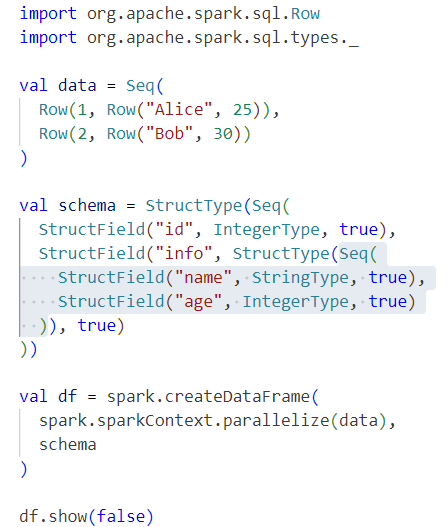
Cuándo y cómo usar:

* **Definir el Esquema de un DataFrame:** Cuando se trabaja con datos sin un esquema explícito (como archivos JSON, CSV o RDDs)



* **Manipulación de Datos Complejos**  
  Puedes manejar columnas que contienen estructuras anidadas (por ejemplo, JSON o mapas) definiendo columnas como StructType dentro de un DataFrame.

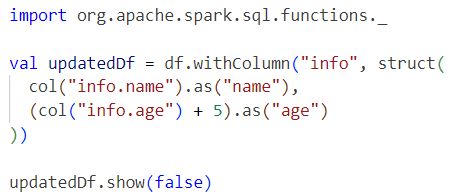
Crear la estructura:

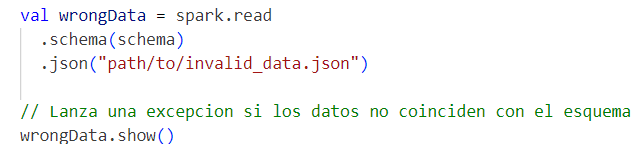


Acceder a los campos de la estructura:



Modificar los campos de la estructura:

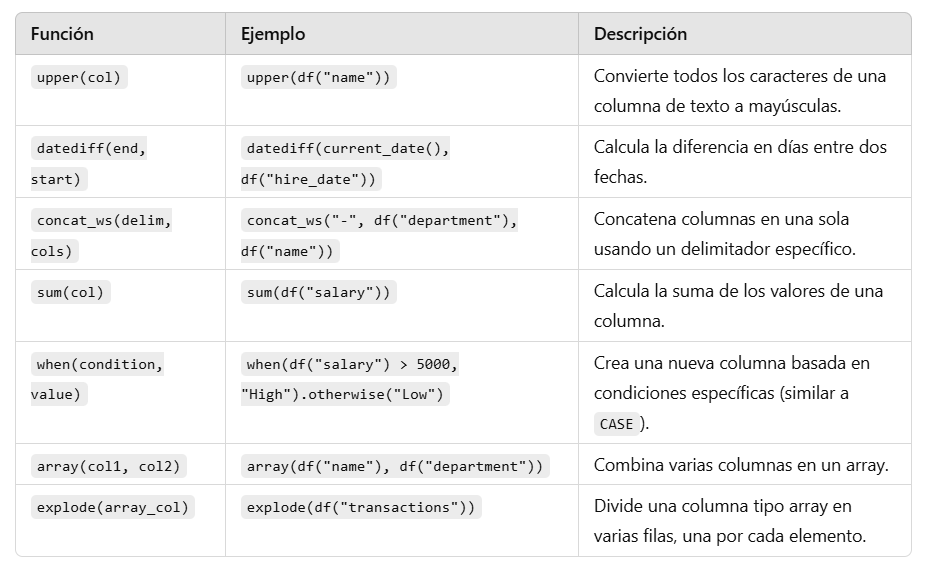


* **Validación de Datos**  
  Un StructType ayuda a Spark a validar que los datos cargados cumplen con el esquema esperado.

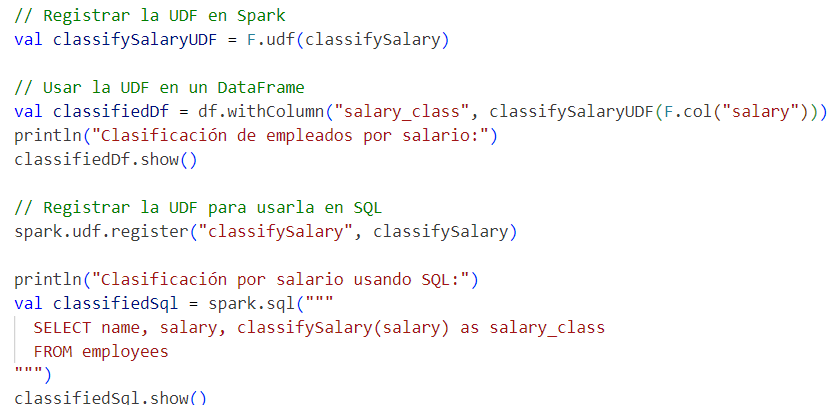
Funciones útiles

Las funciones son fundamentales para transformar y analizar datos en columnas. Estas se dividen en tres categorías principales:

* **Funciones Integradas**: Ofrecidas por Spark (como concat, sum, avg).



* **Funciones Definidas por el Usuario (UDFs):** Personalizadas en Scala para lógica específica.





* **Funciones SQL:** Directamente usadas en consultas SQL.

<https://spark.apache.org/docs/3.5.1/api/sql/index.html>

