

# APACHE SPARK

## Manual de usuario



# Índice

1. ¿Qué es Apache spark?	4
1.1. Introducción a Apache Spark	4
1.2. Características y ventajas	4
1.3. API de Spark	5
1.4. Arquitectura de Spark	6
1.5. Aplicaciones spark	6
1.6. Spark vs hadoop	8
2. Scala	9
2.1. Declaración de variables	9
2.2. Estructuras de control	10
2.3. Declaración de funciones	11
2.4. Tipos complejos	13
2.4.1. Arrays	13
2.4.2. Secuencias	14
2.4.3. Listas	14
2.5. Ejercicios	15
3. Clases de objetos	16
3.1. Ejercicios	18
4. RDD	18
4.1. Creación de un RDD	19
4.2. Transformaciones	20
4.3. Acciones	27
4.4. Ejercicios	29
5. DATAFRAMES	30
6. Dataset	41
7. Rrd vs dataframe vs dataset	42
8. Spark sql	44
8.1. Ejercicios	47
9. Carga y almacenamiento distintos tipos archivos	48
9.1. CSV	48
9.2. JSON	49
9.3. PARQUET	50
10. Spark streaming	50
10.1. Streaming en spark	51

10.2.	Structured streaming.....	51
10.3.	Ejemplo spark streaming .....	52
10.4.	Elementos .....	55
10.5.	Ejercicios .....	58
3.	Bibliografia .....	59

# 1. ¿Qué es Apache spark?

## 1.1. Introducción a Apache Spark

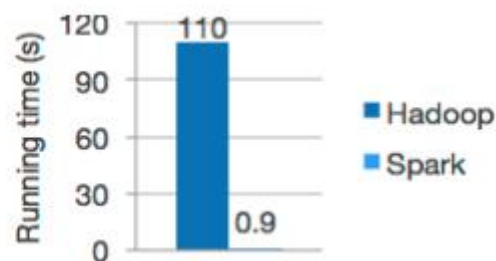
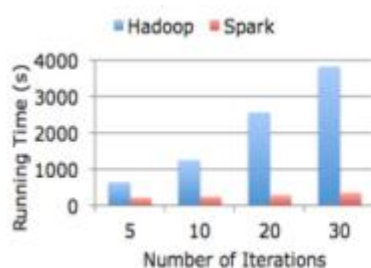
Algunas industrias están utilizando Hadoop para almacenar, procesar y analizar grandes volúmenes de datos. Hadoop se basa en el modelo de programación MapReduce y permite una solución de computación que es escalable, tolerante a fallos, flexible y rentable. La principal preocupación que presenta Hadoop es mantener la velocidad de espera entre las consultas y el tiempo para ejecutar el programa en el procesamiento de grandes conjuntos de datos.

Posteriormente salió a la luz Apache Spark introducido por la empresa Apache Software Foundation para acelerar el proceso de software de cálculo computacional Hadoop. Aunque es importante mencionar que Apache Spark depende de Hadoop, ya que lo utiliza para propósitos de almacenamiento. Apache Spark es una infraestructura informática de clúster de código abierto usado con frecuencia para cargas de trabajo de Big Data1 .

Además, ofrece un desempeño rápido , ya que el almacenamiento de datos se gestiona en memoria, lo que mejora el desempeño de cargas de trabajo interactivas sin costos de E/S (periféricos de entrada/salida). Por otro lado, Apache Spark es compatible con las bases de datos de gráficos, el análisis de transmisiones, el procesamiento general por lotes, las consultas ad-hoc y el aprendizaje automático. Empresas como Alibaba Taobao y Tencent, ya están utilizando Apache Spark como gestor de datos. La empresa Tencent posee actualmente 800 millones de usuarios activos, generando un total de 700 TB de datos procesados al día en un clúster de más de 8000 nodos de computación.

## 1.2. Características y ventajas

- ✚ **Velocidad:** Apache Spark es capaz de ejecutar hasta 100 veces más rápido aplicaciones ejecutadas en memoria y 10 veces más rápido cuando se ejecuta en HDD. Esto se debe principalmente a la reducción de número de operaciones de lectura / escritura en el disco y al nuevo almacenamiento de datos de procesamiento intermedio en memoria.



- ✚ **Potencia:** Apache Spark nos permite realizar más operaciones que Hadoop MapReduce: integración con lenguaje R (Spark R), procesamiento de streaming, cálculo de grafos (GraphX), machine learning (MLlib), y análisis interactivos.
- ✚ **Facilidad de uso:** Uno de los principales problemas que poseía Hadoop, es que requería de usuarios con niveles avanzados de MapReduce o programación

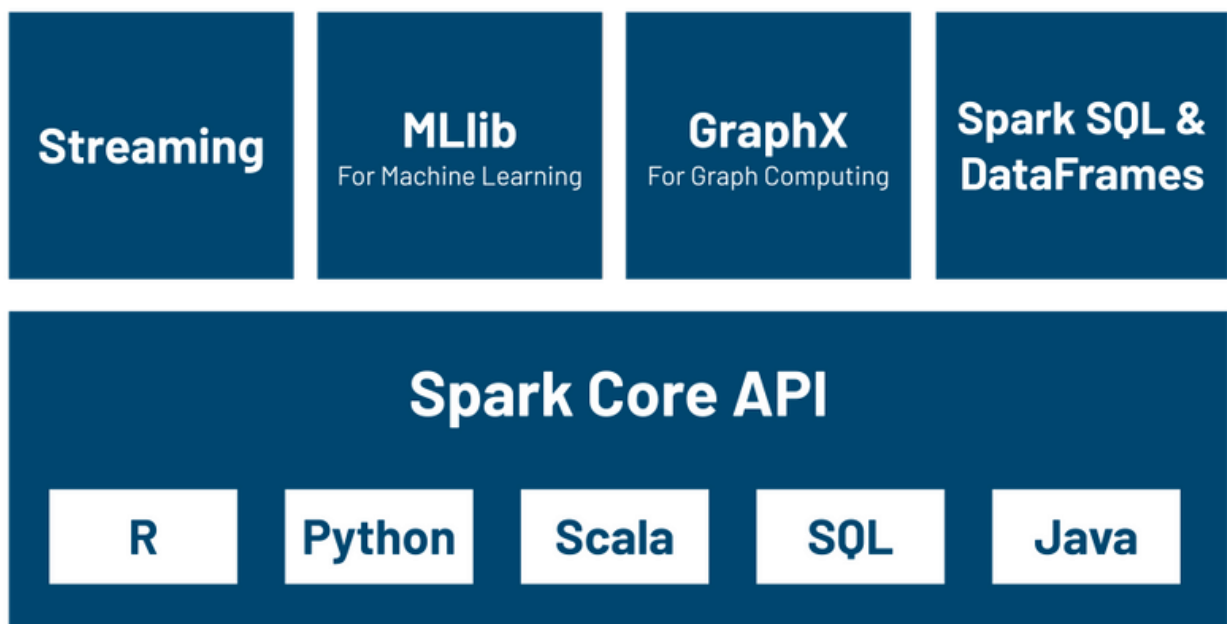
avanzada en Java. Este inconveniente desaparece con la llegada de Spark, ya que gracias a la API nos permite programar en R, Python, Scala e incluso en Java

- + **Entiende sql:** Gracias al módulo Spark SQL se permite la consulta de datos estructurados y semiestructurados utilizando lenguaje SQL o gracias a la API.
- + **Comunidad:** Apache Spark presenta una comunidad cada vez más activa, en la que los desarrolladores mejoran las características de la plataforma, y ayudan al resto de programadores a implementar soluciones o resolver problemas.
- + **Escalabilidad:** Spark nos da la posibilidad de ir incrementando nuestro clúster a medida que vamos necesitando más recursos
- + Procesar y analizar datos que con las tecnologías actuales era imposible

### 1.3. API de Spark

El elemento principal es *Spark Core* el cual aporta toda la funcionalidad necesaria para preparar y ejecutar las aplicaciones distribuidas, gestionando la planificación y tolerancia a fallos de las diferentes tareas. Para ello, el núcleo ofrece un entorno *NoSQL* idóneo para el análisis exploratorio e interactivo de los datos. *Spark* se puede ejecutar en *batch* o en modo interactivo y tiene soporte para *Python*. Independientemente del lenguaje utilizado (ya sea *Python*, *Java*, *Scala*, *R* o *SQL*) el código se despliega entre todos los nodos a lo largo del clúster.

Además, contiene otros 4 grandes componentes contruidos sobre el Core:



- + **Spark Core:** es el corazón de spark, responsable de gestionar las funciones como la programación de las tareas.
- + **Spark Streaming:** Spark es capaz de realizar análisis de streaming sin problemas, gracias a la gran velocidad de programación de su núcleo. Además, gracias a la API se permite crear aplicaciones escalables e intolerantes a fallos de streaming. Otra ventaja que posee Spark, es que es capaz de procesar grandes datos en tiempo real, mientras que MapReduce, solamente es capaz de gestionar datos

en lotes. Gracias a esta ventaja, los datos son analizados conforme entran, sin tiempo de latencia y a través de un proceso de gestión en continuo tránsito.

**Spark SQL** ofrece un interfaz SQL para trabajar con *Spark*, permitiendo la lectura de datos tanto de una tabla de cualquier base de datos relacional como de ficheros con formatos estructurados (CSV, texto, *JSON*, *Avro*, *ORC*, *Parquet*, etc....) y construir tablas permanentes o temporales en *Spark*. Tras la lectura, permite combinar sentencias SQL para trabajar con los datos y cargar los resultados en *DataFrame* de *Spark*.

- ✚ **Spark MLlib** es un módulo de machine learning que ofrece la gran mayoría de algoritmos de ML y permite construir pipelines para el entrenamiento y evaluación de los modelos IA.
- ✚ **GraphX**: Es un entorno de procesamiento gráfico ubicado en la parte superior de *Spark*, el cual proporciona una API para gráficos y cálculo gráfico en paralelo.

## 1.4. Arquitectura de Spark

Hay varios datos útiles que destacar sobre esta arquitectura:

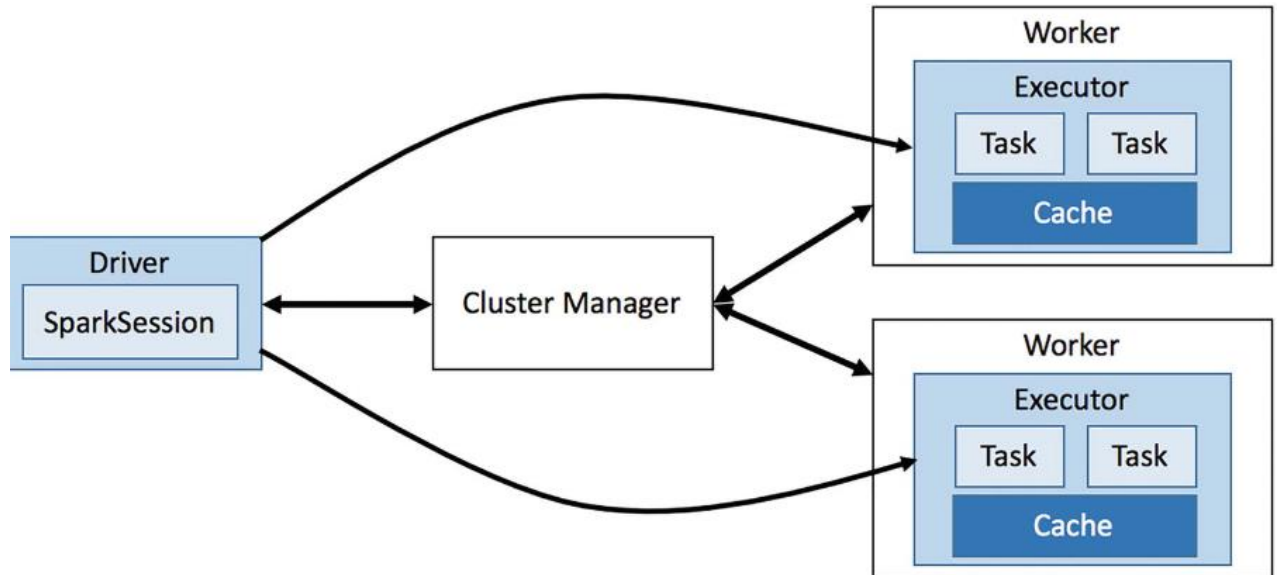
- ✚ Las aplicaciones de *Spark* son ejecutadas independientemente y estas son coordinadas por el objeto *SparkContext* del programa principal (Driver Program4 ).
- ✚ **SparkContext** es capaz de conectarse a gestores de clúster (Cluster Manager), los cuales se encargan de asignar recursos en el sistema. Hay varios tipos de gestores de clúster:
  - **Standalone**: sencillo gestor de clústeres, incluido con *Spark*, que facilita la creación de un clúster.
  - **Apache Mesos**: es un gestor de clústeres un poco más avanzado que el anterior, que puede ejecutar *Hadoop*, *MapReduce* y aplicaciones de servicio.
  - **Hadoop YARN**: es el gestor de recursos en *Hadoop*
- ✚ Una vez conectados, *Spark* puede encargarse de que se creen ejecutores (executors), encargados de ejecutar tareas (tasks) en los nodos del clúster.
- ✚ Los dos componentes principales del clúster son:
  - **Gestor de clúster**: nodo maestro que sabe dónde se localizan los esclavos, cuánta memoria disponen y el número de Cores CPU de cada nodo. Su mayor responsabilidad es orquestar el trabajo asignándolo a los diferentes nodos.
  - **Nodos trabajadores (workers)**: cada nodo ofrece recursos (memoria, CPU, etc....) al gestor del clúster y realiza las tareas que se le asignen.

## 1.5. Aplicaciones spark

Una aplicación *Spark* se compone de dos partes:

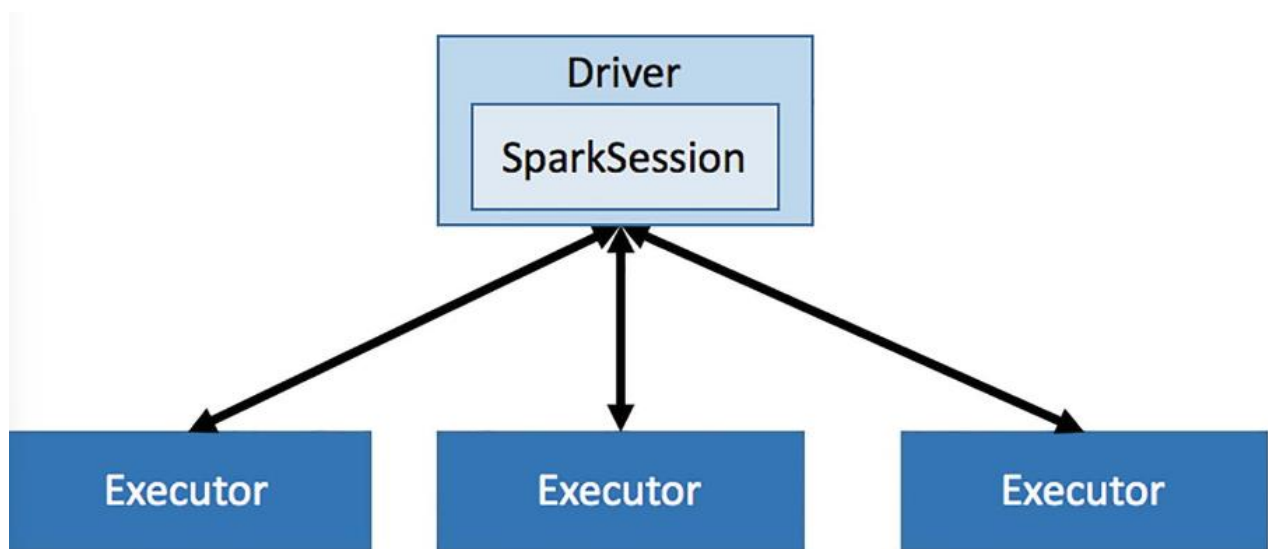
- ✚ La **lógica de procesamiento** de los datos, la cual realizamos mediante alguna de las API que ofrece *Spark* (*Java*, *Scala*, *Python*, etc....), desde algo sencillo que realice una ETL sobre los datos a problemas más complejos que requieran múltiples iteraciones y tarden varias horas como entrenar un modelo de *machine learning*.

- Driver:** es el coordinador central encargado de interactuar con el clúster Spark y averiguar qué máquinas deben ejecutar la lógica de procesamiento. Para cada una de esas máquinas, el driver realiza una petición al clúster para lanzar un proceso conocido como ejecutor (executor). Además, el driver Spark es responsable de gestionar y distribuir las tareas a cada ejecutor, y si es necesario, recoger y fusionar los datos resultantes para presentarlos al usuario. Estas tareas se realizan a través de la SparkSession.






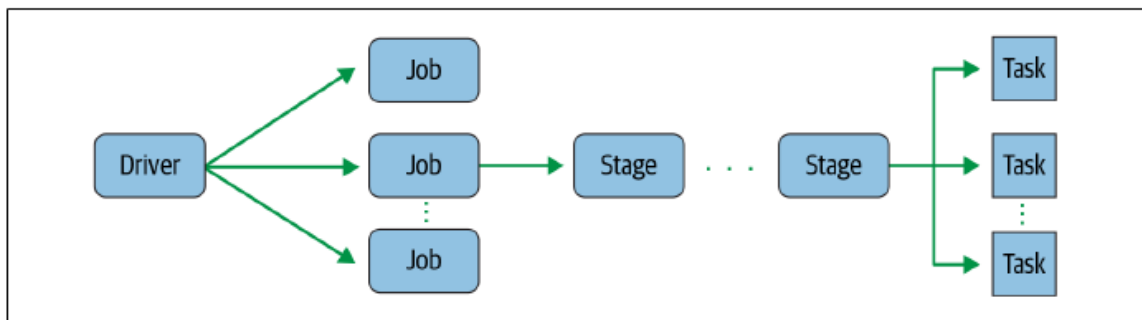
Arquitectura entre una aplicación Spark y el gestor del clúster

Así pues, *Spark* utiliza una arquitectura maestra/esclavo, donde el *driver* es el maestro, y los ejecutores los esclavos. Cada uno de estos componentes se ejecutan como un proceso independiente en el clúster *Spark*. Por lo tanto, una aplicación *Spark* se compone de un *driver* y múltiples ejecutores. Al lanzar una aplicación *Spark*, podemos indicar el número de ejecutores que necesita la aplicación, así como la cantidad de memoria y número de núcleos que debería tener cada ejecutor.



Cuando creamos una aplicación Spark, por debajo, se distinguen los siguientes elementos:

-  **Job** (trabajo): computación paralela compuesta de múltiples tareas que se crean tras una acción de Spark (save, collect, etc...). Al codificar nuestro código mediante PySpark, el driver convierte la aplicación Spark en uno o más jobs, y a continuación, estos jobs los transforma en un DAG (grafo). Este grafo, en esencia, es el plan de ejecución, donde cada elemento dentro del DAG puede implicar una o varias stages (escenas).
-  **Stage** (escena): cada Job se divide en pequeños conjuntos de tareas que forman un escenario. Como parte del grafo, las stages se crean a partir de si las operaciones se pueden realizar de forma paralela o de forma secuencial. Como no todas las operaciones pueden realizarse en una única stage, en ocasiones se dividen en varias, normalmente debido a los límites computacionales de los diferentes ejecutores.
-  **Task** (tarea): unidad de trabajo más pequeña que se envía a los ejecutores Spark. Cada escenario se compone de varias tareas. Cada una de las tareas se asigna a un único núcleo y trabaja con una única partición de los datos. Por ello, un ejecutor con 16 núcleos puede tener 16 o más tareas trabajando en 16 o más particiones en paralelo.



## 1.6. Spark vs hadoop

No es fácil hacer una comparación entre Hadoop y Spark, ya que ambos realizan cosas similares, aunque poseen algunas áreas donde sus funcionalidades no se superponen. Por ejemplo, Spark no posee un sistema de archivos y, por ello, debe apoyarse en el sistema de archivos de Hadoop (HDFS).

	SPARK	HADOOP
<b>RENDIMIENTO</b>	Trabaja con memoria → acelera procesos	Trabaja con el disco → procesos más lentos
	Hace falta memoria para almacenar datos	Almacenamiento inferior a spark
	Rendimiento afectado con aplicaciones pesadas	Elimina datos cuando no son necesarios → no se produce pérdida de rendimiento
<b>USABILIDAD</b>	Mas facilidad al usar la API	Mas complejo



<b>COSTES</b>	Necesita tanta memoria como datos a tratar.	Opción más barata ya que almacena todo en disco
	Más rentable con menos hardware se realizan más tareas	Necesita más hardware de procesamiento
<b>COMPATIBILIDAD</b>	Compatible con mapreduce	Compatible con spark
	Posee compatibilidades con mapreduce para fuentes de datos	
<b>PROCESAMIENTO DATOS</b>	Más operaciones a parte del procesamiento de datos en una sola plataforma	Necesita más plataformas, ideneo para procesamiento en lotes.
<b>TOLERANCIA A FALLOS</b>	Si un proceso se bloquea debería comenzar de nuevo	Debido a que procesa en disco si falla se puede comenzar donde se dejó.
<b>SEGURIDAD</b>	Necesita Hadoop YARN para obtener beneficios de seguridad	Posee YARN
	Necesita HDFS para acceder a permisos de nivel de archivo	Posee level Authorization
		Autenticación por kerberos

## 2. Scala

### 2.1. Declaración de variables

🚦 Variable mutable:

```
var miVariable: Int = 10
```

🚦 Variable Inmutable (constante):

```
val miConstante: String = "Hola, mundo!"
```

🚦 Inferencia de tipo

```
val miNumero = 42 // Scala infiere que es un Int
```

## 2.2. Estructuras de control

### Condicional If/else

La estructura de control if se utiliza para tomar decisiones basadas en condiciones

```
if (condicion) {  
  instrucciones  
} else {  
  instrucciones  
}
```

Ejemplo:

```
val numero = 10  
  
if (numero > 5) {  
  println("El número es mayor que 5")  
} else {  
  println("El número no es mayor que 5")  
}
```

### Bucle for

El bucle **for** se usa para iterar sobre colecciones de datos, como arrays, listas o rangos de números.

```
for (variable <- secuencia) {  
  // Cuerpo del bucle  
  // ...  
}
```

Ejemplo:

```
var list = List(1,2,3,4,5,6)  
for (n <- list)  
  println(n)
```

### Bucle foreach

La estructura de bucle **foreach** se utiliza para iterar sobre elementos de una colección sin la necesidad de escribir explícitamente un bucle for:

```
coleccion.foreach { elemento =>  
  // Cuerpo del bucle  
  // Hacer algo con el 'elemento'  
}
```

Ejemplo:

```
val miLista = List("A", "B", "C")

miLista.foreach { elemento =>
  println(elemento)
}
```

## Bucle foryield

El bucle for con yield se utiliza para construir nuevas colecciones aplicando transformaciones a los elementos de una colección existente. La estructura básica es similar a un bucle for estándar, pero se utiliza la palabra clave yield para construir una nueva colección a partir de los resultados de la iteración.

```
val mi_array = Array(1,2,3,4,5)
for (x <- mi_array) yield x * 2

res: Array[Int] = Array(2, 4, 6, 8, 10)
```

## Bucle while

El bucle while se utiliza para repetir un bloque de código mientras una condición sea verdadera.

```
while (condicion) {
  // Cuerpo del bucle
  // ...
}
```

Ejemplo:

```
var contador = 0

while (contador < 5) {
  println(s"Contador: $contador")
  contador += 1
}
```

## 2.3. Declaración de funciones

### Función sin parámetros

Las funciones sin parámetros se definen con la siguiente sintaxis:

```
def miFuncion(): TipoDeRetorno = {
  // Cuerpo de la función
  // ...
}
```

Ejemplo:

```
def saludar(): String = {  
    "¡Hola, mundo!"  
}  
  
// Llamada a la función  
val mensaje = saludar()  
println(mensaje)
```

## Función con parámetros

Las funciones con parámetros se definen con la siguiente sintaxis:

```
def miFuncion(parametro1: Tipo1, parametro2: Tipo2, ...): TipoDeRetorno = {  
    // Cuerpo de la función  
    // ...  
}
```

Ejemplo:

```
def sumar(a: Int, b: Int): Int = {  
    a + b  
}  
  
// Llamada a la función  
val resultado = sumar(3, 5)  
println(resultado)
```

## Función con numero de parámetros variable

Se puede definir una función con un número variable de parámetros utilizando el operador \*. Este operador permite que la función acepte cero o más argumentos del tipo especificado

```
def miFuncion(parametro1: Tipo1, parametrosRestantes: TipoRestante*): TipoDeRetorno = {  
    // Cuerpo de la función  
    // ...  
}
```

Ejemplo:

```
def sumarNumeros(entero1: Int, numerosRestantes: Int*): Int = {  
    val sumaRestantes = numerosRestantes.sum  
    entero1 + sumaRestantes  
}
```

```
// Llamada a la función con varios argumentos
val resultado1 = sumarNumeros(1, 2, 3, 4, 5)
val resultado2 = sumarNumeros(10, 20, 30)

println(resultado1) // Imprime 15
println(resultado2) // Imprime 60
```

## Función anónima

Las funciones anónimas se conocen como funciones literales o funciones lambda.

```
val miFuncionAnonima = (parametro1: Tipo1, parametro2: Tipo2, ...) => {
    // Cuerpo de la función
    // ...
}
```

Ejemplo:

```
val sumar = (a: Int, b: Int) => a + b

// Llamada a la función anónima
val resultado = sumar(3, 5)
println(resultado)
```

## 2.4. Tipos complejos

### 2.4.1. Arrays

Los arrays son colecciones mutables de elementos. Lo que significa que puedes modificarlos después de crearlos.

**Crear un array:**

```
// Crear un array de enteros
val miArray: Array[Int] = Array(1, 2, 3, 4, 5)

// Crear un array de strings
val miArrayDeStrings: Array[String] = Array("Hola", "Mundo")

// Crear un array vacío de longitud 3
val arrayVacio: Array[Int] = new Array[Int](3)
```

**Acceder a sus elementos:**

```
/ Acceder a un elemento por índice
val primerElemento: Int = miArray(0)

// Modificar un elemento
miArray(1) = 10
```

**Iterar sobre un array:**

```
// Iterar sobre elementos utilizando un bucle for
for (elemento <- miArray) {
  println(elemento)
}
```

```
// Otra forma utilizando foreach
miArray.foreach { elemento =>
  println(elemento)
}
```

### Operaciones comunes

```
// Longitud del array
val longitud: Int = miArray.length

// Sumar todos los elementos del array
val suma: Int = miArray.sum

// Encontrar el máximo y mínimo
val maximo: Int = miArray.max
val minimo: Int = miArray.min
```

## 2.4.2. Secuencias

En Scala, puedes utilizar secuencias (también conocidas como listas o colecciones) en Apache Spark para trabajar con datos distribuidos. Sin embargo, a diferencia de las secuencias regulares en Scala, las secuencias en Spark se representan como Resilient Distributed Datasets (RDDs) o DataFrames (que veremos más adelante).

La sintaxis para trabajar con secuencias en Spark es similar a trabajar con otras estructuras de datos, como arrays o listas.

## 2.4.3. Listas

Las listas son una estructura de datos inmutable y genérica. Se pueden crear listas con elementos del mismo tipo o de diferentes tipos.

### Declaración de listas

```
// Lista de enteros
val listaEnteros: List[Int] = List(1, 2, 3, 4, 5)

// Lista de cadenas
val listaCadenas: List[String] = List("Hola", "Mundo", "Scala")

// Lista vacía
val listaVacía: List[Nothing] = List()

// Lista de cualquier tipo (no se recomienda a menos que sea necesario)
val listaMixta: List[Any] = List(1, "dos", 3.0)
```

## Transformar columna de dataframe en lista

```
val df = Seq(
  ("uno", 2.0),
  ("dos", 1.5),
  ("tres", 8.0)
).toDF("id", "valor")

val lista = df.select("id").map(r => r.getString(0)).collect.toList

res: List[String] = List(uno, dos, tres)
```

## Añadir elementos a una lista

```
val nuevaLista1 = numeros :+ 6 // Agrega al final
val nuevaLista2 = 0 +: numeros // Agrega al principio
val nuevaLista3 = numeros ++ List(7, 8, 9) // Concatena listas
```

## Filtros sobre listas

```
list1.filter(_ > 2)
list1.filter(_ % 2 == 0)
res1: List[Int] = List(3, 4, 5)
res2: List[Int] = List(2, 4)
```

## 2.5. Ejercicios

1. Crea dos variables a y b con los valores 1 y 5 respectivamente. A continuación, comprueba e indica cuál de ellos es mayor y cual es menor.
2. Crea un bucle for para recorrer los números del 1 al 10. A continuación realiza la suma de los números pares y la suma de los números impares y muéstralo por pantalla.
3. Crear un método llamado ObtenerCuadrado que realice el cuadrado de un numero double que ingreses. A continuación, prueba el método con las siguientes sentencias.

```
val sd1 = ObtenerCuadrado (1.2)
assert(1.44 == sd1, "mensaje ")
val sd2 = ObtenerCuadrado (5.7)
assert(32.49 == sd2, "mensaje")
```

4. crear un método llamado isArg1 GreaterThanArg2 en le que introduzcas dos double. Retorna true si el primer argumento es mayor y en caso contrario false. A continuación pruebalo con la siguiente sentencia.

```
val t1 = isArg1 GreaterThanArg2(4.1, 4.12)
assert(/* fill this in */)
val t2 = isArg1 GreaterThanArg2(2.1, 1.2)
assert(/* fill this in */)
```

5. Crear un método GetMe que recibe un String y devuelva ese string en minúsculas. A continuación, llama al método e imprime el resultado.
6. Crea un método que usando un bucle while verifiques si el numero introducido es primo o no.
7. crear un método en el que introduzcas un numero variable de argumentos de números double y se realice la multiplicación de todos ellos
8. Implementa un método llamado sumarEnteros que reciba una lista de enteros y los sume, devuelve el resultado de la suma.
9. implementa un método llamado AnalizarNumeros que reciba una lista de enteros y diga cuales son > 0 cuales <0 y cuántos 0

### 3. Clases de objetos

#### Declaración clases de objetos

Una clase es una plantilla para la creación de objetos. En Scala, puedes definir una clase de la siguiente manera:

```
class Persona(nombre: String, edad: Int) {  
  def saludar(): Unit = {  
    println(s"Hola, mi nombre es $nombre y tengo $edad años.")  
  }  
}  
  
var o1 = new Persona("hola",15);  
o1.saludar();
```

En este ejemplo, hemos definido una clase Persona con dos parámetros en el constructor (nombre y edad). También hay un método saludar que imprime un saludo con el nombre y la edad de la persona.

Sin embargo, los atributos nombre y edad no están accesibles para modificarlos fuera de la clase (estarían como private) si queremos hacer que sean accesibles desde fuera de la clase tendríamos que declararlos con *var* o *val*.

```
class Persona(var nombre: String,var edad: Int) {  
  def saludar(): Unit = {  
    println(s"Hola, mi nombre es $nombre y tengo $edad años.")  
  }  
}  
  
var o1 = new Persona("ainhoa",15);  
  
Println(o1.nombre);
```

#### Case class

Todavía hay una cantidad significativa de código repetitivo al crear clases que



principalmente contienen datos y nos los modifican. Scala intenta eliminar la repetición siempre que puede, y eso es lo que hace la clase de case. Esta clase se define de la siguiente manera

```
case class NombreTipo(arg1:Tipo, arg2:Tipo, ...)
```

A primera vista, parece una clase normal con la palabra clave delante. Sin embargo, una clase de case crea automáticamente todos los argumentos de clase como vals. Si necesitas en su lugar, un argumento de clase es una var, basta con colocar una var delante del argumento.

En spark esta clase se utiliza para trabajar con datos inmutables como son los datasets y dataframes.

En el presente ejemplo:

- Definimos una case class llamada Persona con dos campos (nombre y edad).
- Creamos instancias de la case class utilizando su constructor.
- Accedemos a los campos de la case class.
- Utilizamos la case class en el contexto de Spark para crear un DataFrame.

```
// Definición de una case class
case class Persona(nombre: String, edad: Int)

// Crear una instancia de la case class
val persona1 = Persona("Juan", 25)
val persona2 = Persona("Ana", 30)

// Acceder a los campos de la case class
println(s"Nombre: ${persona1.nombre}, Edad: ${persona1.edad}")

// Utilizar con Spark DataFrame
import org.apache.spark.sql.{SparkSession, DataFrame}

// Crear una sesión de Spark
val spark = SparkSession.builder.appName("EjemploCaseClass").master("local[*]").getOrCreate()

// Crear un DataFrame a partir de una secuencia de case class
val personasSeq = Seq(Persona("Juan", 25), Persona("Ana", 30), Persona("Luis", 28))
val personasDF: DataFrame = spark.createDataFrame(personasSeq)

// Mostrar el DataFrame
personasDF.show()
```

Cuando se utiliza una **case class** en Spark, se facilita la creación de **DataFrames** y **Datasets**, ya que automáticamente obtienes funcionalidades como la generación automática de métodos **hashCode**, **equals**, y **toString**, lo cual es útil para operaciones como el join y el agrupamiento. Además, las **case class** son inmutables, lo que significa que sus instancias no pueden cambiar después de ser creadas, lo cual es beneficioso para trabajar con datos inmutables en un contexto distribuido como Spark.

### 3.1. Ejercicios

1. crea una clase rectángulo con dos argumentos : anchura y altura y los siguientes métodos:

- calcular area: ancho \* altura
- calcular perímetro: (anchura+altura) \* 2
- imprimir detalles

A continuación, pruébala con las siguientes sentencias.

```
// Crear instancias de la clase Rectangulo
val rectangulo1 = new Rectangulo(5.0, 3.0)
val rectangulo2 = new Rectangulo(7.0, 2.0)

// Imprimir detalles de los rectángulos
rectangulo1.imprimirDetalles()
println("-----")
rectangulo2.imprimirDetalles()
```

## 4. RDD

Un RDD (**Resilient Distributed Datasets**) es una estructura de datos que abstrae los datos para su procesamiento en paralelo.

Antes de *Spark* 2.0, los RDD eran el interfaz principal para interactuar con los datos.

Se trata de una colección de elementos tolerantes a fallos que son inmutables (una vez creados, no se pueden modificar) y diseñados para su procesamiento distribuido. Cada conjunto de datos en los RDD se divide en particiones lógicas, que se pueden calcular en diferentes nodos del clúster.

Hay dos formas de crear un RDD:

- Paralelizando una colección ya existente en nuestra aplicación *Spark*.
- Referenciando un dataset de un sistema externo como *HDFS*, *HBase*, etc...

Sobre los RDD se pueden realizar dos tipos de operaciones:

- **Acción:** devuelven un valor tras ejecutar una computación sobre el conjunto de datos.
- **Transformación:** es una operación perezosa que crea un nuevo conjunto de datos a partir de otro RDD/Dataset, tras realizar un filtrado, *join*, etc...

## 4.1. Creación de un RDD

### Usando Parallelize

Podemos crear RDD directamente desde cero sin necesidad de leer los datos desde un fichero. Para ello, a partir de un [SparkContext](#) podemos utilizar [parallelize](#) sobre una lista.

Esta acción divide una colección de elementos entre los nodos de nuestro clústers. Por ejemplo:

```
val miRDD = sc.parallelize(Array(1, 2, 3, 4, 5, 6, 7, 8, 9))

// Crear RDD a partir de una lista de Strings
val lista = Array("Hola", "Adiós", "Hasta luego")
val listaRDD = sc.parallelize(lista)

// Crear RDD a partir de una lista de Strings con 4 particiones
val listaRDD4 = sc.parallelize(lista, 4)
```

### Leyendo un archivo de texto

```
val rutaArchivo = "/tmp/sample.txt" // Cambia esto según tu configuración

// Crear un RDD a partir del archivo de texto
val rddDesdeArchivo = sc.textFile(rutaArchivo)

// Mostrar las primeras 5 líneas del RDD para verificar la carga
rddDesdeArchivo.take(5).foreach(println)
```

Ojo! La ruta del archivo de texto puede variar dependiendo la fuente de datos.  
En el ejemplo se muestra una fuente de datos local

### Leer csv

```
// Ruta del archivo CSV en HDFS o en el sistema de archivos local
val rutaCSV = "/tmp/mnm_dataset.csv" // Ajusta según tu configuración

// Crear un RDD a partir de las líneas del archivo CSV
val rdd = sc.textFile(rutaCSV)

// Mostrar las primeras 5 líneas del RDD
rdd.take(5).foreach(println)
```

### Desde un dataframe o un dataset ya existentes

Para convertir DataSet o DataFrame a RDD se realiza usando la función **rdd()**. Ten en cuenta que convertir un DataFrame o DataSet a un RDD puede tener implicaciones de

rendimiento, y generalmente se prefiere trabajar directamente con DataFrames o DataSets cuando sea posible, ya que ofrecen optimizaciones y una interfaz más rica para manipulación de datos en Spark.

```
// Crear un DataFrame de ejemplo
val df = spark.createDataFrame(Seq((1, "A"), (2, "B"), (3, "C"))).toDF("id", "value")

// Obtener un RDD a partir del DataFrame
val rddFromDF = df.rdd

// Mostrar las primeras 5 filas del RDD
rddFromDF.take(5).foreach(println)

import org.apache.spark.sql.{SparkSession, Dataset}

import spark.implicits._
import org.apache.spark.rdd

case class Persona(id: Int, nombre: String, edad: Int)

// Crear un DataSet a partir de una secuencia de datos
val datosSeq = Seq(Persona(1, "Alice", 25), Persona(2, "Bob", 30), Persona(3, "Charlie", 22))
val dataset = spark.createDataset(datosSeq)

// Mostrar el contenido del DataSet
dataset.show()

val rdd1: org.apache.spark.rdd.RDD[Persona] = dataset.rdd
```

## 4.2. Transformaciones

En *Spark*, las estructuras de datos son inmutables, de manera que una vez creadas no se pueden modificar. Para poder modificar un *RDD/DataFrame*, hace falta realizar una transformación, siendo el modo de expresar la lógica de negocio mediante *Spark*.

Todas las transformaciones en *Spark* se evalúan de manera perezosa (**lazy evaluation**), de manera que los resultados no se computan inmediatamente, sino que se retrasa el cálculo hasta que el valor sea necesario. Para ello, se van almacenando los pasos necesarios y se ejecutan únicamente cuando una acción requiere devolver un resultado al *driver*. Este diseño facilita un mejor rendimiento (por ejemplo, imagina que tras una operación *map* se realiza un *reduce* y en vez de devolver todo el conjunto de datos tras el *map*, sólo le enviamos al *driver* el resultado de la reducción).

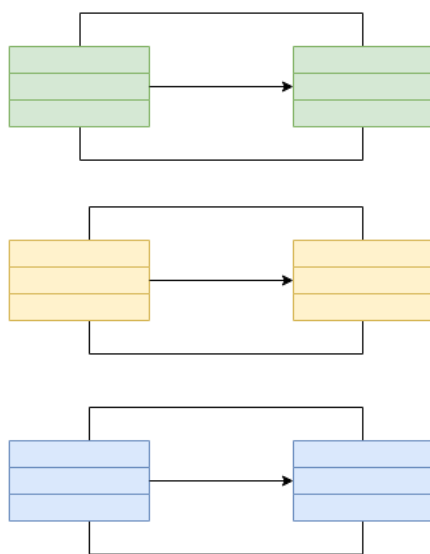
Así pues, las acciones provocan la evaluación de todas las transformaciones previas que se habían evaluado de forma perezosa y estaban a la espera.

Por defecto, cada transformación *RDD/DataSet* se puede recalcularse cada vez que se ejecute una acción. Sin embargo, podemos persistir un *RDD* en memoria mediante los métodos `persist` (o `cache`), de manera que *Spark* mantendrá los datos para un posterior acceso más eficiente. También podemos persistir *RDD* en disco o replicarlo en múltiples nodos.

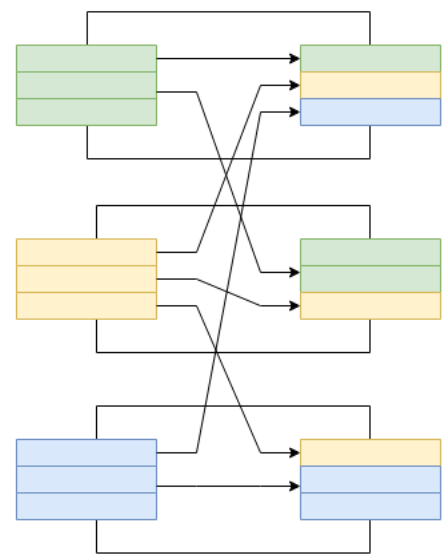
## Tipos de transformaciones¶

Existen dos tipos de transformaciones, dependiendo de las dependencias entre las particiones de datos:

- Transformaciones **Narrow**: consisten en dependencias *estrechas* en las que cada partición de entrada contribuye a una única partición de salida.
- Transformaciones **Wide**: consisten en dependencias *anchas* de manera que varias particiones de entrada contribuyen a muchas otras particiones de salida, es decir, cada partición de salida depende de diferentes particiones de entrada. Este proceso también se conoce como *shuffle*, ya que *Spark* baraja los datos entre las particiones del clúster.



Transformación narrow



Transformación wide

Con las transformaciones *narrow*, *Spark* realiza un *pipeline* de las dependencias, de manera que si especificamos múltiples filtros sobre DataFrames/RDD, se realizarán todos en memoria.

Esto no sucede con las transformaciones *wide*, ya que al realizar un *shuffle* los resultados se persisten .

### !!!! Cuidado con shuffle !!

Las operaciones *shuffle* son computacionalmente caras, ya que implican E/S en disco, serialización de datos y E/S en red. Para organizar los datos previos al *shuffle*, *Spark* genera un conjunto de tareas (tareas *map* para organizar los datos, y *reduce* para agregar los resultados).

Internamente, el resultado de las tareas *map* se mantienen en memoria hasta que no caben. Entonces, se ordenan en la partición destino y se persisten en un único archivo. En la fase de reducción, las tareas leen los bloques ordenados que son relevantes.

Las operaciones *reduceByKey* y *aggregateByKey* son de las que más memoria consumen, al tener que crear las estructuras de datos para organizar los registros en las tareas de *map*, y luego generar los resultados agregados en la de *reduce*. Si los datos no caben en memoria, *Spark* los lleva a disco, incurriendo en operaciones adicionales de E/S en disco y del recolector de basura.

## RESUMEN

Transformación	Definición	Ejemplo
<b>map(func)</b>	Devuelve un nuevo RDD tras pasar cada elemento del RDD original a través de una función.	<pre>val v1 = sc.parallelize(List(2, 4, 8)) val v2 = v1.map(_ * 2) v2.collect res0: Array[Int] = Array(4, 8, 16)</pre>
<b>filter(func)</b>	Realiza un filtrado de los elementos del RDD original para devolver un nuevo RDD con los datos filtrados.	<pre>val v1 = sc.parallelize(List("ABC", "BCD", "DEF")) val v2 = v1.filter(_.contains("C")) v2.collect res0: Array[String] = Array(ABC)</pre>
<b>flatMap(func)</b>	Parecido a la operación <i>map</i> , pero la función devuelve una secuencia de valores.	<pre>val x = sc.parallelize(List("Ejemplo proyecto Alejandro", "Hola mundo"), 2) val y = x.map(x =&gt; x.split(" ")) y.collect res0: Array[Array[String]] = Array(Array(Ejemplo, proyecto, Alejandro), Array(Hola, mundo)) val y = x.flatMap(x =&gt; x.split(" ")) y.collect res1: Array[String] = Array(Ejemplo, proyecto, Alejandro, Hola, mundo)</pre>
<b>mapPartitions(func)</b>	Similar a la operación <i>map</i> , pero se ejecuta por separado en cada	<pre>val a = sc.parallelize(1 to 9, 3) def myfunc[T](iter: Iterator[T]) : Iterator[(T, T)] = {   var res = List[(T, T)]()   var pre = iter.next   while (iter.hasNext)</pre>

partición del RDD.

```
{val cur = iter.next;
res ::= (pre, cur)
pre = cur;}
res.iterator}
a.mapPartitions(myfunc).collect
```

```
res0: Array[(Int, Int)] = Array((2,3), (1,2), (5,6), (4,5), (8,9), (7,8))
```

### **sample(withReplacement, fraction, seed)**

Muestra una fracción de los datos, con o sin reemplazo, utilizando una semilla que genera números aleatorios.

```
val randRDD = sc.parallelize(List( (7,"cat"), (6,"mouse"),(7, "cup"), (6,"book"), (7, "tv"), (6, "screen"),(7,"heater")))
val sampleMap = List((7, 0.4), (6, 0.6)).toMap
randRDD.sampleByKey(false,sampleMap,42).collect
```

```
res41: Array[(Int, String)] = Array((6,book), (7,tv), (7,heater))
```

### **union(otherDataset)**

Devuelve un nuevo RDD con la unión de los elementos de los RDDs seleccionados

```
val a = sc.parallelize(1 to 3, 1)
val b = sc.parallelize(5 to 7, 1)
a.union(b).collect()
```

```
res42: Array[Int] = Array(1, 2, 3, 5, 6, 7)
```

### **intersection(otherDataset)**

Devuelve los elementos de los RDDs que son iguales.

```
val x = sc.parallelize(1 to 20)
val y = sc.parallelize(10 to 30)
val z = x.intersection(y) z.collect
```

```
res0: Array[Int] = Array(16, 14, 12, 18, 20, 10, 13, 19, 15, 11, 17)
```

### **distinct([numTasks])**

Devuelve los elementos de los RDDs que son distintos.

```
val c = sc.parallelize(List("Gnu", "Cat", "Rat", "Dog","Gnu", "Rat"), 2)
c.distinct.collect
```

```
res0: Array[String] = Array(Dog, Gnu, Cat, Rat)
```

### **groupByKey([numTasks])**

Similar al grupoBy, realiza el agrupamiento por clave de un conjunto de datos, pero en lugar de suministrar una función,

```
a = sc.parallelize(List("dog", "tiger", "lion", "cat","spider", "eagle"), 2)
val b = a.keyBy(_.length)
b.groupByKey.collect
```

```
res45: Array[(Int, Iterable[String])] =
Array((4,CompactBuffer(lion)), (6,CompactBuffer(spider)), (3,CompactBuffer(cat, dog)), (5,CompactBuffer(eagle, tiger)))
```

el  
componente  
clave de  
cada par se  
presentará  
automática  
mente al  
particionador  
.

### **reduceByKey(func, [numTasks])**

Devuelve un  
conjunto de  
datos de  
pares (K, V)  
donde los  
valores de  
cada clave  
son  
agregados  
usando la  
función de  
reducción  
dada

```
val a = sc.parallelize(List("dog", "cat", "owl", "gnu", "ant"), 2)
val b = a.map(x => (x.length, x))
b.reduceByKey(_ + _).collect

res0: Array[(Int, String)] = Array((3,dogcatowlgnuant))
```

### **aggregateByKey(z er oValue)(seqOp, comb Op, [numTasks])**

Devuelve un  
conjunto de  
datos de  
pares (K, U)  
donde los  
valores de  
cada clave  
se agregan  
utilizando las  
funciones  
combinadas  
dadas y un  
valor por  
defecto de:  
"cero".

```
val nombres = sc.parallelize(List(("David", 6), ("Abby", 4),
  ("David", 5), ("Abby", 5)))
nombres.aggregateByKey(0)((k,v) => v.toInt+k, (v,k) =>
  k+v).collect

res0: Array[(String, Int)] = Array((Abby,9), (David,11))
```

### **sortByKey([ascendi n g], [numTasks])**

Esta función  
ordena los  
datos del  
RDD de  
entrada y los  
almacena en  
un nuevo  
RDD.

```
val a = sc.parallelize(List("dog", "cat", "owl", "gnu", "ant"), 2)
val b = sc.parallelize(1 to a.count.toInt, 2)
val c = a.zip(b)
c.sortByKey(true).collect

res48: Array[(String, Int)] = Array((ant,5), (cat,2), (dog,1),
  (gnu,4), (owl,3))

c.sortByKey(false).collect

res49: Array[(String, Int)] = Array((owl,3), (gnu,4), (dog,1),
  (cat,2), (ant,5))
```



<b>join(otherDataset, numTasks)</b>	Realiza una unión interna utilizando dos RDD de valor clave. Cuando se introduce conjuntos de datos de tipo (K, V) y (K, W), se devuelve un conjunto de datos de (K, (V, W)) para cada clave	<pre>val a = sc.parallelize(List("dog", "salmon", "salmon", "rat", "elephant"), 3) val b = a.keyBy(_.length) val c = sc.parallelize(List("dog","cat","gnu","salmon","rabbit","turkey", "wolf","bear","bee"), 3) val d = c.keyBy(_.length) b.join(d).collect  res50: Array[(Int, (String, String))] = Array((6,(salmon,salmon)), (6,(salmon,rabbit)), (6,(salmon,turkey)), (6,(salmon,salmon)), (6,(salmon,rabbit)), (6,(salmon,turkey)), (3,(dog,dog)), (3,(dog,cat)), (3,(dog,gnu)), (3,(dog,bee)), (3,(rat,dog)), (3,(rat,cat)), (3,(rat,gnu)), (3,(rat,bee)))</pre>
<b>cogroup(otherDataset, numTasks)</b>	Un conjunto muy potente de funciones que permiten agrupar hasta tres valores claves de RDDs utilizando sus claves.	<pre>val a = sc.parallelize(List(1, 2, 1, 3), 1) val b = a.map((_, "b")) val c = a.map((_, "c")) val d = a.map((_, "d")) b.cogroup(c, d).collect  res0: Array[(Int, (Iterable[String], Iterable[String], Iterable[String]))] = Array( (2,(ArrayBuffer(b),ArrayBuffer(c),ArrayBuffer(d))), (3,(ArrayBuffer(b),ArrayBuffer(c),ArrayBuffer(d))), (1,(ArrayBuffer(b, b),ArrayBuffer(c, c),ArrayBuffer(d, d))) )</pre>
<b>cartesian(otherDataset)</b>	Calcula el producto cartesiano entre dos RDD, es decir cada elemento del primer RDD se une a cada elemento del segundo RDD, y los devuelve como un nuevo RDD.	<pre>val x = sc.parallelize(List(1,2,3,4,5)) val y = sc.parallelize(List(6,7,8,9,10)) x.cartesian(y).collect  res0: Array[(Int, Int)] = Array((1,6), (1,7), (1,8), (1,9), (1,10), (2,6), (2,7), (2,8), (2,9), (2,10), (3,6), (3,7), (3,8), (3,9), (3,10), (4,6), (5,6), (4,7), (5,7), (4,8), (5,8), (4,9), (4,10), (5,9), (5,10))</pre>
<b>pipe(command, envVars)</b>	Toma los datos RDD de cada partición y los envía a través de stdin a un	<pre>val a = sc.parallelize(1 to 9, 3) a.pipe("head -n 1").collect  res0: Array[String] = Array(1, 4, 7)</pre>

shell-  
command

<b>coalesce(numPartitions)</b>	Disminuye el número de particiones en el RDD al número especificado ( numPartitions )	<pre>val y = sc.parallelize(1 to 10, 10) val z = y.coalesce(2, false) z.partitions.length res0: Int = 2</pre>
<b>repartition(numPartitions)</b>	Reorganiza aleatoriamente los datos en el RDD para crear más o menos particiones.	<pre>val x = (1 to 12).toList val numbersDf = x.toDF("number") numbersDf.rdd.partitions.size res0: Int = 4 Partition 00000: 1, 2, 3 Partition 00001: 4, 5, 6 Partition 00002: 7, 8, 9 Partition 00003: 10, 11, 12 val numbersDfR = numbersDf.repartition(2) Partition A: 1, 3, 4, 6, 7, 9, 10, 12 Partition B: 2, 5, 8, 11</pre>
<b>repartitionAndSortWithinPartitions(partitioner)</b>	Reparte el RDD de acuerdo con el particionador dado y, dentro de cada partición resultante, clasifica los registros por sus claves. Como se puede observar en el ejemplo pedimos que los datos sean organizados en dos particiones: A y C como una partición y, B y D como otra.	<pre>&gt;&gt; pairs = sc.parallelize([["a",1], ["b",2], ["c",3], ["d",3]]) &gt;&gt; pairs.collect() # Output [['a', 1], ['b', 2], ['c', 3], ['d', 3]] &gt;&gt; pairs.repartitionAndSortWithinPartitions(2).glom().collect() # Output [['a', 1], ('c', 3)], [('b', 2), ('d', 3)]] // Reorganización basado en cierta condición. &gt;&gt; pairs.repartitionAndSortWithinPartitions(2,partitionFunc=lambda x: x == 'a').glom().collect() # Output [('b', 2), ('c', 3), ('d', 3)], [('a', 1)]]</pre>

## 4.3. Acciones

Acción	Definición	Ejemplo
<b>reduce(func)</b>	Agrega los elementos del dataset usando una función. Esta función debe ser conmutativa y asociativa para que pueda calcularse correctamente en paralelo	<pre>val a = sc.parallelize(1 to 100, 3) a.reduce(_ + _) res0: Int = 5050</pre>
<b>collect()</b>	Convierte un RDD en un array y lo muestra por pantalla	<pre>val c = sc.parallelize(List("Gnu", "Cat", "Rat", "Dog", "Gnu", "Rat"), 2) c.collect res0: Array[String] = Array(Gnu, Cat, Rat, Dog, Gnu, Rat)</pre>
<b>count()</b>	Devuelve el número de elementos del dataset.	<pre>val a = sc.parallelize(1 to 4) a.count res0: Long = 4</pre>
<b>first()</b>	Devuelve el primer elemento del conjunto de datos	<pre>val c = sc.parallelize(List("Gnu", "Cat", "Rat", "Dog"), 2) c.first res0: String = Gnu</pre>
<b>take(n)</b>	Devuelve un array con los primeros n elementos del dataset	<pre>val b = sc.parallelize(List("dog", "cat", "ape", "salmon", "gnu"), 2) b.take(2) res0: Array[String] = Array(dog, cat)</pre>
<b>takeSample(withReplacement, n, um, [seed])</b>	Devuelve un array con una muestra aleatoria de elementos numéricos del dataset, con o sin sustitución, con la opción de especificar opcionalmente una semilla de generador de números aleatorios.	<pre>val x = sc.parallelize(1 to 200, 3) x.takeSample(true, 20, 1) res0: Array[Int] = Array(74, 164, 160, 41, 123, 27, 134, 5, 22, 185, 129, 107, 140, 191, 187, 26, 55, 186, 181, 60)</pre>
<b>takeOrdered(n, [ordering])</b>	Devuelve los primeros n elementos del RDD usando su orden original o	<pre>val b = sc.parallelize(List("dog", "cat", "ape", "salmon", "gnu"), 2) b.takeOrdered(2)</pre>

	un comparador personalizado.	<code>res0: Array[String] = Array(ape, cat)</code>
<b>saveAsTextFile(path)</b>	Guarda el RDD como un archivo de texto.	<pre>val a = sc.parallelize(1 to 10000, 3) a.saveAsTextFile("/home/usuario/datos") root@master:/home/usuario/datos # ls part-00000 part-00001 part00002 _SUCCESS  // Como se puede observar se han creado las 3 particiones, las cuales hemos especificado.</pre>
<b>saveAsSequenceFile(path)</b>	Guarda el RDD como un archivo de secuencia Hadoop.	<pre>val v = sc.parallelize(Array(("owl",3), ("gnu",4), ("dog",1), ("cat",2), ("ant",5)), 2) v.saveAsSequenceFile("/home/usuario/seq_datos") root@master:/home/usuario/seq_datos# ls -a .. part-00000 .part-00000.crc part-00001 .part-00001.crc _SUCCESS _SUCCESS.crc</pre>
<b>saveAsObjectFile(path) (Java and Scala)</b>	Guarda los elementos del conjunto de datos en un formato simple utilizando la serialización de Java	<pre>val x = sc.parallelize(Array(("owl",3), ("gnu",4), ("dog",1), ("cat",2), ("ant",5)), 2) x.saveAsObjectFile("/home/usuario/objFile") root@master:/home/usuario/objFile # ls -a . .. part-00000 .part-00000.crc part-00001 .part-00001.crc _SUCCESS _SUCCESS.crc</pre>
<b>countByKey()</b>	Sólo disponible en RDD de tipo (K, V). Devuelve un hashmap de pares (K, Int) con el recuento de cada clave.	<pre>val c = sc.parallelize(List((3, "Gnu"), (3, "Yak"), (5, "Mouse"), (3, "Dog")), 2) c.countByKey res3: scala.collection.Map[Int,Long] = Map(3 -&gt; 3, 5 -&gt; 1)</pre>
<b>foreach(func)</b>	Ejecute una función func en cada elemento del dataset.	<pre>val c = sc.parallelize(List("cat", "dog", "tiger", "lion", "gnu", "crocodile", "ant", "whale", "dolphin", "spider"), 3) c.foreach(x =&gt; println(x + "s are beautiful")) cats are beautiful dogs are beautiful tigers are beautiful ants are beautiful whales are beautiful dolphins are beautiful spiders are beautiful lions are beautiful gnus are beautiful crocodiles are beautiful</pre>

## 4.4. Ejercicios

1. Si tenemos dos RDD (A y B):

```
val rddA = sc.parallelize(Array(1, 2, 3, 4))  
val rddB = sc.parallelize(Array(3, 4, 5, 6))
```

¿Cómo conseguimos los elementos que están en A y no B y los de B que no están en A? (es decir [1, 2, 5, 6])

2. A partir de la lista

siguiente ['Alicante', 'Elche', 'Valencia', 'Madrid', 'Barcelona', 'Bilbao', 'Sevilla']:

- Almacena sólo las ciudades que tengan la letra e en su nombre y muéstralas.
- Muestra las ciudades que tienen la letra e y el número de veces que aparece en cada nombre. Por ejemplo ('Elche', 2).
- Averigua las ciudades que solo tengan una única e.
- Nos han enviado una nueva lista pero no han separado bien las ciudades. Reorganiza la lista y colocalas correctamente, y cuenta las apariciones de la letra e de cada ciudad.

ciudades\_mal =

[['Alicante.Elche', 'Valencia', 'Madrid.Barcelona', 'Bilbao.Sevilla'], ['Murcia', 'San Sebastián', 'Melilla.Aspe']]

3. A partir de las siguientes listas:

Inglés: hello, table, angel, cat, dog, animal, chocolate, dark, doctor, hospital, computer

Español: hola, mesa, angel, gato, perro, animal, chocolate, oscuro, doctor, hospital, ordenador

Una vez creado un RDD con tuplas de palabras y su traducción (puedes usar zip para unir dos listas):

[('hello', 'hola'), ('table', 'mesa'), ('angel', 'angel'), ('cat', 'gato')]...

Averigua:

- Palabras que se escriben igual en inglés y en español
  - Palabras que en español son distintas que en inglés
  - Obtén una única lista con las palabras en ambos idiomas que son distintas entre ellas ([ 'hello', 'hola', 'table', ...])
  - Haz dos grupos con todas las palabras, uno con las que empiezan por vocal y otro con las que empiecen por consonante.
4. Dada una cadena que contiene una lista de nombres Juan, Jimena, Luis, Cristian, Laura, Lorena, Cristina, Jacobo, Jorge, una vez transformada la cadena en una lista y luego en un RDD:
- Agrúpalos según su inicial, de manera que tengamos tuplas formadas por la letra inicial y todos los nombres que comienzan por dicha letra:

```
('J', ['Juan', 'Jimena', 'Jacobo', 'Jorge']),  
( 'L', ['Luis', 'Laura', 'Lorena']),  
( 'C', ['Cristian', 'Cristina'])
```

- De la lista original, obtén una muestra de 5 elementos sin repetir valores.

- Devuelve una muestra de datos de aproximadamente la mitad de los registros que la lista original con datos que pudieran llegar a repetirse.
5. Dada una lista de elementos desordenados y algunos repetidos, devolver una muestra de 5 elementos, que estén en la lista, sin repetir y ordenados descendientemente.

```
lista = 4,6,34,7,9,2,3,4,4,21,4,6,8,9,7,8,5,4,3,22,34,56,98
```

- Selecciona el elemento mayor de la lista resultante.
  - Muestra los dos elementos menores.
6. En una red social sobre cine, tenemos un fichero ratings.txt compuesta por el código de la película, el código del usuario, la calificación asignada y el timestamp de la votación con el siguiente formato:

```
1::1193::5::978300760  
1::661::3::978302109  
1::914::3::978301968
```

Se pide crear código usando rdd para:

- Obtener para cada película, la nota media de todas sus votaciones.
  - Películas cuya nota media sea superior a 3.
- ```
lista = 4,6,34,7,9,2,3,4,4,21,4,6,8,9,7,8,5,4,3,22,34,56,98
```
- Selecciona el elemento mayor de la lista resultante.
  - Muestra los dos elementos menores.
7. Tenemos las calificaciones de las asignaturas de matemáticas, inglés y física de los alumnos del instituto en 3 documentos de texto. A partir de estos ficheros:
- Crea 3 RDD de pares, uno para cada asignatura, con los alumnos y sus notas
  - Crea un solo RDD con todas las notas
  - ¿Cuál es la nota más baja que ha tenido cada alumno?
  - ¿Cuál es la nota media de cada alumno?
  - ¿Cuántos estudiantes suspende cada asignatura?  
[('Mates', 7), ('Física', 8), ('Inglés', 7)]
  - ¿En qué asignatura suspende más gente?
  - Total de notables o sobresalientes por alumno, es decir, cantidad de notas superiores o igual a 7.
  - ¿Qué alumno no se ha presentado a inglés?
  - ¿A cuántas asignaturas se ha presentado cada alumno?
  - Obtén un RDD con cada alumno con sus notas

## 5. DATAFRAMES

Un *DataFrame* es una estructura equivalente a una tabla de base de datos relacional, con un motor bien optimizado para el trabajo en un clúster. Los datos se almacenan en filas y columnas y ofrece un conjunto de operaciones para manipular los datos.

El trabajo con *DataFrames* es más sencillo y eficiente que el procesamiento con RDD, por eso su uso es predominante en los nuevos desarrollos con *Spark*.

A continuación veremos cómo podemos obtener y persistir *DataFrames* desde diferentes fuentes y formatos de datos

## 5.1. Crear un dataframe

La forma más simple de crear un dataframe es a través de un RDD como se muestra en el ejemplo.

```
val data = List(
  Row("Paco","Garcia",24,24000),
  Row("Juan","Garcia",26,27000),
  Row("Lola","Martin",29,31000),
  Row("Sara","Garcia",35,34000)
)
val rdd = sc.parallelize(data)
val schema = StructType(
  List(
    StructField("nombre", StringType, nullable=false),
    StructField("apellido", StringType, nullable=false),
    StructField("edad", IntegerType),
    StructField("salario", IntegerType)
  )
)
val df = spark.createDataFrame(rdd,schema)
df.printSchema()
df.show()
```

Otra forma de crear un dataframe es mediante el uso de un esquema.

Un esquema en Spark define los nombres de las columnas y los tipos de datos asociados para un DataFrame. En la mayoría de los casos, los esquemas entran en juego cuando se lee datos estructurados desde una fuente de datos externa (más sobre esto en el próximo capítulo). Definir un esquema de antemano, en lugar de adoptar un enfoque de esquema sobre la lectura, ofrece tres beneficios:

- Libera a Spark de la carga de inferir los tipos de datos.
- Evita que Spark cree un trabajo separado solo para leer una gran parte de su archivo y determinar el esquema, lo cual puede ser costoso y llevar mucho tiempo para un archivo de datos grande.
- Permite detectar errores temprano si los datos no coinciden con el esquema.

Por lo tanto, recomendamos que siempre definas su esquema de antemano cuando desees leer un archivo grande desde una fuente de datos.

Spark te permite definir un esquema. Una es definirlo programáticamente, y la otra es emplear una cadena de lenguaje de definición de datos (DDL), que es mucho más simple y fácil de leer.

Para crear un dataframe mediante su API se hace de la siguiente manera:

```
import org.apache.spark.sql.{SparkSession, Row}
import org.apache.spark.sql.types.{StructType, StructField, StringType, IntegerType}

val schema = StructType(
  Array(
    StructField("Nombre", StringType, nullable = true),
    StructField("Edad", IntegerType, nullable = true)
  )
)

// Crear filas de datos
val data = Seq(
  Row("Juan", 15),
  Row("María", 30),
  Row("Carlos", 28)
)

// Crear un RDD a partir de las filas
val rdd = spark.sparkContext.parallelize(data)

// Crear el DataFrame
val df = spark.createDataFrame(rdd, schema)

// Mostrar el DataFrame
df.show()
```

## 5.2. Columnas

Como se mencionó anteriormente, las columnas con nombres en los DataFrames son conceptualmente similares a una tabla de un sistema de gestión de bases de datos relacional (RDBMS): describen un tipo de campo.

Puedes enumerar todas las columnas por sus nombres y realizar operaciones en sus valores utilizando expresiones relacionales o computacionales. En los lenguajes compatibles con Spark, las columnas son objetos con métodos públicos (representados por el tipo de dato Columna).



También puedes utilizar expresiones lógicas o matemáticas en las columnas. Por ejemplo, podrías crear una expresión simple usando `expr("nombreColumna * 5")` o `(expr("nombreColumna - 5") > col(otroNombreColumna))`, donde `nombreColumna` es un tipo de dato Spark (entero, cadena, etc.). `expr()` forma parte de los paquetes `pyspark.sql.functions` (Python) y `org.apache.spark.sql.functions` (Scala). Al igual que cualquier otra función en esos paquetes, `expr()` toma argumentos que Spark interpretará como una expresión, calculando el resultado.

Por ejemplo sobre el ejemplo del dataframe anterior podríamos multiplicar la edad de cada persona por 2 de la siguiente manera:

```
df.select(expr("Edad * 2")).show()
```

o se podría añadir una nueva columna que indicara si la persona es mayor de edad o no de la siguiente forma:

```
df.withColumn("Mayor edad", (expr("Edad > 18"))).show()
```

para más información sobre columnas consultar la [API de spark](#)

### 5.3. Rows

En Spark, una fila (Row) es un objeto genérico que contiene una o más columnas. Cada columna puede ser del mismo tipo de datos (por ejemplo, entero o cadena) o pueden tener tipos diferentes (entero, cadena, mapa, array, etc.). Dado que Row es un objeto en Spark y una colección ordenada de campos, puedes instanciar una Row en cada uno de los lenguajes compatibles con Spark y acceder a sus campos mediante un índice que comienza en 0.

Los objetos Row pueden utilizarse para crear DataFrames si los necesitas para una interactividad y exploración rápidas como hemos hecho en el anterior ejemplo.

### 5.4. Operaciones

Para realizar operaciones comunes en DataFrames, primero necesitarás cargar un DataFrame desde una fuente de datos que contenga tus datos estructurados. Spark proporciona una interfaz, `DataFrameReader`, que te permite leer datos en un DataFrame desde una variedad de fuentes de datos en formatos como JSON, CSV, Parquet, Texto, Avro, ORC, etc. De manera similar, para escribir un DataFrame de nuevo en una fuente de datos en un formato particular, Spark utiliza `DataFrameWriter`.

#### Lectura y escritura de un dataframe

La lectura y escritura son simples en Spark gracias a estas abstracciones de alto nivel y a las contribuciones de la comunidad para conectarse a una amplia variedad de fuentes de datos, que incluyen almacenes NoSQL comunes, sistemas de gestión de bases de datos relacionales (RDBMS), motores de transmisión como Apache Kafka y Kinesis, y más.

| Método | Argumentos | Descripción |
|--------|------------|-------------|
|--------|------------|-------------|

|                 |                                                                                                                                |                                                                                                                                                                                                                                                                               |
|-----------------|--------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Format()</b> | "parquet", "csv", "txt", "json", "jdbc", "orc", "avro", etc.                                                                   | Si no especificas este método, entonces el valor predeterminado es Parquet o lo que esté configurado en spark.sql.sources.default.                                                                                                                                            |
| <b>Option()</b> | ("mode", {PERMISSIVE   FAILFAST   DROPMALFORMED } )<br>("inferSchema", {true   false})<br>("path",<br>"path_file_data_source") | Una serie de pares clave/valor y opciones. La documentación de Spark muestra algunos ejemplos y explica los diferentes modos y sus acciones. El modo predeterminado es PERMISSIVE. Las opciones "inferSchema" y "mode" son específicas de los formatos de archivo JSON y CSV. |
| <b>Schema()</b> | DDL String or StructType, e.g., 'A INT, B STRING' or StructType(...)                                                           | Para el formato JSON o CSV, puedes especificar la inferencia del esquema utilizando el método option(). En general, proporcionar un esquema para cualquier formato acelera la carga y garantiza que tus datos cumplan con el esquema esperado.                                |
| <b>Load()</b>   | "/path/to/data/source"                                                                                                         | La ruta hacia la fuente de datos. Esto puede estar vacío si se especifica en option("path", "...").                                                                                                                                                                           |

Por ejemplo para leer un csv y con el esquema (si no se especifica esquema spark lo inferirá por ti.

```
import org.apache.spark.sql.{SparkSession, Row}
import org.apache.spark.sql.types.{StructType, StructField, StringType, IntegerType, BooleanType, FloatType}

val fireSchema = StructType(Array(
  StructField("CallNumber", IntegerType, true), StructField("UnitID", StringType, true),
  StructField("IncidentNumber", IntegerType, true), StructField("CallType", StringType, true),
  StructField("CallDate", StringType, true), StructField("WatchDate", StringType, true),
  StructField("CallFinalDisposition", StringType, true), StructField("AvailableDtTm", StringType, true),
  StructField("Address", StringType, true), StructField("City", StringType, true),
  StructField("Zipcode", IntegerType, true), StructField("Battalion", StringType, true),
  StructField("StationArea", StringType, true), StructField("Box", StringType, true),
  StructField("OriginalPriority", StringType, true), StructField("Priority", StringType, true),
  StructField("FinalPriority", IntegerType, true), StructField("ALSUnit", BooleanType, true),
  StructField("CallTypeGroup", StringType, true), StructField("NumAlarms", IntegerType, true),
  StructField("UnitType", StringType, true), StructField("UnitSequenceInCallDispatch", IntegerType, true),
  StructField("FirePreventionDistrict", StringType, true), StructField("SupervisorDistrict", StringType, true),
  StructField("Neighborhood", StringType, true), StructField("Location", StringType, true),
  StructField("RowID", StringType, true), StructField("Delay", FloatType, true) ))

val sfFireFile="/tmp/sf-fire-calls.csv"
val fireDF = spark.read.schema(fireSchema).option("header", "true").csv(sfFireFile)
```

Se puede guardar el dataframe en un archivo con los siguientes métodos:

| Método               | Argumentos                                                                                                                                                                                     | Descripción                                                                                                                                                                                                                                                                   |
|----------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Format()</b>      | "parquet", "csv", "txt", "json", "jdbc", "orc", "avro", etc.                                                                                                                                   | Si no especificas este método, entonces el valor predeterminado es Parquet o lo que esté configurado en spark.sql.sources.default.                                                                                                                                            |
| <b>Option()</b>      | ("mode", {append   overwrite   ignore   error or errorifexists} )<br>("mode", {SaveMode.Overwrite   SaveMode.Append, SaveMode.Ignore, SaveMode.ErrorIfExists})<br>("path", "path_to_write_to") | Una serie de pares clave/valor y opciones. La documentación de Spark muestra algunos ejemplos y explica los diferentes modos y sus acciones. El modo predeterminado es PERMISSIVE. Las opciones "inferSchema" y "mode" son específicas de los formatos de archivo JSON y CSV. |
| <b>bucketBy()</b>    | (numBuckets, col, col..., coln)                                                                                                                                                                | El número de buckets y los nombres de las columnas por los cuales agrupar. Utiliza el esquema de bucketing de Hive en un sistema de archivos.                                                                                                                                 |
| <b>save()</b>        | "/path/to/data/source"                                                                                                                                                                         | La ruta hacia la fuente de datos. Esto puede estar vacío si se especifica en option("path", "...").                                                                                                                                                                           |
| <b>saveAsTable()</b> | "table_name"                                                                                                                                                                                   |                                                                                                                                                                                                                                                                               |

Por ejemplo:

```
val Path = "/tmp/pruebas.csv"
fireDF.write.format("csv").save(Path)
```

## Operaciones

### Mostrar datos

Para mostrar los datos, ya hemos visto que podemos utilizar el método show, al cual le podemos indicar o no la cantidad de registros a recuperar, así como si queremos que los datos se trunquen o no, o si los queremos mostrar en vertical:

```
fireDF.show()
// indicandole cuantos queremos mostrar
fireDF.show(2)
//truncando
fireDF.show(truncate=3)
//mostrando la tabla en vertical
fireDF.show(numRows = 3, truncate = 50, vertical = true)
```

Si sólo queremos recuperar unos pocos datos, podemos hacer uso de head o first los cuales devuelven objetos Row:

```
fireDF.first()
fireDF.head()
fireDF.head(3)
```

Si queremos obtener un valor en concreto, una vez recuperada una fila, podemos acceder a sus columnas:

```
val nom1 = fireDF.first().get(0)
val nom2 = fireDF.first().getAs[String]("City")
```

También podemos obtener un resumen de los datos mediante describe:

```
fireDF.describe.show()
```

Si únicamente nos interesa saber cuántas filas tiene nuestro DataFrame, podemos hacer uso de count:

```
fireDF.count
```

Por último, como un DataFrame por debajo es un RDD, podemos usar collect y take conforme necesitemos y recuperar objetos de tipo Row:

```
fireDF.collect
fireDF.take(2)
```

## Select

la operación select permite indicar las columnas a recuperar pasándolas como parámetros:

```
fireDF.select(fireDF("City")).show(2)
```

También podemos realizar cálculos (referenciando a los campos con nombreDataFrame.nombreColumna) sobre las columnas y crear un alias (operación asociada a un campo):

```
fireDF.select(col("UnitID"), (fireDF("CallNumber") + 10).alias("CallNumberMas10")).show(3)
```

Si tenemos un DataFrame con un gran número de columnas y queremos recuperarlas todas a excepción de unas pocas, es más cómodo utilizar la transformación drop, la cual funciona de manera opuesta a select, es decir, indicando las columnas que queremos quitar del resultado:

```
fireDF.drop("UnitID", "CallNumber").show(5)
```

Una vez tenemos un DataFrame, podemos añadir columnas mediante el método withColumn:

```
fireDF.withColumn("CallNumbermas10", col("CallNumber") * 10)
```

Otra forma de añadir una columna con una expresión es mediante la transformación selectExpr. Por ejemplo, podemos conseguir el mismo resultado que en el ejemplo anterior de la siguiente manera:

```
fireDF.selectExpr("*", "CallNumber * 10").show()
```

Aunque más adelante veremos cómo realizar transformaciones con agregaciones, mediante `selectExpr` también podemos realizar analítica de datos aprovechando la potencia de SQL:

```
fireDF.selectExpr("count(distinct(city)) as cities").show()
```

Si por algún extraño motivo necesitamos cambiarle el nombre a una columna (por ejemplo, vamos a unir dos DataFrames que tienen columnas con el mismo nombre pero en posiciones diferentes, o que al inferir el esquema tenga un nombre críptico o demasiado largo y queremos que sea más legible) podemos utilizar la transformación `withColumnRenamed`:

```
fireDF.withColumnRenamed("IncidentNumber", "IncidentNumber1111111").show(5)
```

## Filtrar

Para filtrar usaremos el método `filter`

```
fireDF.filter(col("City") != "SF").show()
```

Un caso particular de filtrado es la eliminación de los registros repetidos, lo cual lo podemos hacer de dos maneras

```
fireDF.select("City").distinct().show()
```

```
fireDF.dropDuplicates(Seq("City")).select("City").show()
```

## Ordenar

Una vez recuperados los datos deseados, podemos ordenarlos mediante `sort` u `orderBy` (son operaciones totalmente equivalentes):

```
// Selección de "City" y "Location", ordenadas por "Location"
```

```
fireDF.select("City", "Location").sort("Location").show(5)
```

```
// Ordenación del DataFrame completo por "City" en ascendente
```

```
fireDF.sort("City").show(5)
```

```
// Ordenación del DataFrame completo por "City" en orden descendente
```

```
fireDF.orderBy(col("City").desc).show(5)
```

```
// Ordenación por "City" en orden descendente y por "Location" en orden ascendente
```

```
fireDF.sort(col("City").desc, col("Location").asc).show(5)
```

## Trabajar con nulos

Si queremos saber si una columna contiene nulos, podemos hacer un filtrado utilizando el método `isNull` sobre los campos deseados (también podemos utilizar `isNotNull` si queremos el caso contrario):

```
fireDF.filter(col("City").isNull()).count()
```

## Agregaciones

Una vez tenemos un DataFrame, podemos realizar analítica de datos sobre el dataset entero, o sobre una o más columnas y aplicar una función de agregación que permita sumar, contar o calcular la media de cualquier grupo, entre otras opciones.

Para ello, scala ofrece un amplio conjunto de funciones. En nuestro caso, vamos a realizar algunos ejemplos para practicar con las funciones más empleadas.

### Contar

Devuelve la cantidad de elementos no nulos

```
fireDF.filter(col("City").isNull()).count()
```

### Calcular (min, max, sum, avg)

```
fireDF.select(min("CallDate"), max("Priority")).show()
```

```
fireDF.select(sum("NumAlarms")).show()
```

```
fireDF.select(avg("NumAlarms")).show()
```

### Agrupar

Si agrupamos varias columnas de tipo categóricas (con una cardinalidad baja), podemos realizar cálculos sobre el resto de las columnas.

Sobre un DataFrame, podemos agrupar los datos por la columna que queramos utilizando el método `groupBy`, el cual nos devuelve un `GroupedData`, sobre el que posteriormente realizar operaciones como `avg(cols)`, `count()`, `mean(cols)`, `min(cols)`, `max(cols)` o `sum(cols)`:

```
fireDF.groupBy("City").count().show()
```

Si necesitamos realizar más de una agregación sobre el mismo grupo, mediante `agg` podemos indicar una o más expresiones de columnas:

```
fireDF.groupBy("City").agg(sum("NumAlarms").as("Total"),count("Location").as("CountLoc")).show()
```

## Funciones

Para dominar realmente Spark, hay que tener destreza en todas las funciones existente para el tratamiento de fechas, cadenas, operaciones matemáticas, para trabajar con colecciones, etc...

[Mas información](#)

## Fechas

- Si necesitamos convertir de texto a fecha: `to_date`, `to_timestamp`, `unix_timestamp`
- Si necesitamos convertir de texto a fecha: `to_date`, `to_timestamp`, `unix_timestamp`
- Si necesitamos convertir de texto a fecha: `to_date`, `to_timestamp`, `unix_timestamp`
- Si necesitamos convertir de texto a fecha: `to_date`, `to_timestamp`, `unix_timestamp`

## Cadenas

- Por ejemplo, tenemos las funciones para quitar espacios (`ltrim`, `rtrim`, `trim`) y pasar a mayúsculas/minúsculas (`lower`, `upper`).
- O funciones para poner la inicial en mayúsculas (`initcap`), darle la vuelta (`reverse`), obtener su tamaño (`length`) o reemplazar caracteres (`translate`).
- También podemos trabajar con subcadenas (`substring`), encontrar ocurrencias (`locate`) o partir una cadena en trozos (`split`).
- Otras funciones que se suelen utilizar son `concat` y `concat_ws` para unir cadenas, `levenshtein` para calcular la distancia entre dos cadenas, `lpad` y `rpadd` para completar con espacios, etc... Si necesitas trabajar con expresiones regulares puedes utilizar `regexp_extract` para extraer parte de una cadena como `regexp_replace` para sustituir.

## 5.5. Ejercicios

1. A partir del archivo [nombres.json](#), crea un *DataFrame* y realiza las siguientes operaciones:
  - a. Crea una nueva columna (columna Mayor30) que indique si la persona es mayor de 30 años.
  - b. Crea una nueva columna (columna FaltanJubilacion) que calcule cuantos años le faltan para jubilarse (supongamos que se jubila a los 67 años)
  - c. Crea una nueva columna (columna Apellidos) que contenga XYZ (puedes utilizar la función `lit`)
  - d. Elimina las columnas Mayor30 y Apellidos.
  - e. Crea una nueva columna (columna AnyoNac) con el año de nacimiento de cada persona (puedes utilizar la función `current_date`).
  - f. Añade un id incremental para cada fila (campo Id) y haz que al hacer un `show` se vea en primer lugar (puedes utilizar la función `monotonically_increasing_id`) seguidos del Nombre, Edad, AnyoNac, FaltaJubilacion y Ciudad

Al realizar los seis pasos, el resultado del *DataFrame* será similar a :

```
+---+-----+---+-----+-----+-----+
| Id|Nombre|Edad|AnyoNac|FaltanJubilacion| Ciudad|
+---+-----+---+-----+-----+-----+
0	Aitor	45	1977	22	Elche
1	Marina	14	2008	53	Alicante
2	Laura	19	2003	48	Elche
3	Sonia	45	1977	22	Aspe
4	Pedro	null	null	null	Elche
```

+---+-----+---+-----+-----+-----+

2. A partir del archivo VentasNulos.csv:

- a. Elimina las filas que tengan al menos 4 nulos.
- b. Con las filas restantes, sustituye:
  - i. Los nombres nulos por Empleado
  - ii. Las ventas nulas por la media de las ventas de los compañeros (redondeado a entero).
  - iii. Los euros nulos por el valor del compañero que menos € ha ganado. (tras agrupar, puedes usar la función min)
  - iv. La ciudad nula por C.V. y el identificador nulo por XYZ

Para los pasos ii) y iii) puedes crear un *DataFrame* que obtenga el valor a asignar y luego pasarlo como parámetro al método para rellenar los nulos.

3. A partir del archivo movies.tsv, crea un esquema de forma declarativa con los campos:

- Interprete de tipo string
- película de tipo string
- año de tipo int

Cada fila del fichero implica que el actor/actriz ha trabajado en dicha película en el año indicado.

Una vez creado el esquema, carga los datos en un *DataFrame*.

A continuación, mediante el *DataFrame* API:

- Muestra las películas en las que ha trabajado Murphy, Eddie (I).
- Muestra los intérpretes que aparecen tanto en Superman como en Superman II.

4. Sobre las películas de la sesión anterior:

- a. ¿Cuántas películas diferentes hay?
- b. ¿En cuantas películas ha trabajado Murphy, Eddie (I)?
- c. ¿Cuáles son los actores que han aparecido en más de 30 películas?
- d. ¿En qué película anterior a 1980 aparecen al menos 25 intérpretes?
- e. Muestra la cantidad de películas producidas cada año (solo debe mostrar el año y la cantidad), ordenando el listado por la cantidad de forma descendente.
- f. A partir de la consulta anterior, crea un gráfico de barras que muestre el año y la cantidad de películas, ordenados por fecha.

5. Entre los autores de este libro hay una científica de datos que ama hornear galletas con M&Ms, y premia a sus estudiantes en los estados de EE. UU. donde



imparte con frecuencia cursos de aprendizaje automático y ciencia de datos con lotes de esas galletas. Pero ella sigue un enfoque basado en datos, obviamente, y quiere asegurarse de que obtenga los colores correctos de M&Ms en las galletas para los estudiantes en los diferentes estados.

Escribamos un programa de Spark que lea un archivo con más de 100,000 entradas (donde cada fila o línea tiene un <estado, color\_mnm, recuento>) y calcule y agregue los recuentos para cada color y estado. Estos recuentos agregados nos dicen los colores de M&Ms preferidos por los estudiantes en cada estado. El listado completo se encuentra en el csv "mnm\_dataset.csv".

## 6. Dataset

Los Datasets ofrecen una API unificada y singular para objetos fuertemente tipados. Entre los lenguajes admitidos por Spark, solo Scala y Java son fuertemente tipados; por lo tanto, Python y R admiten solo la API de DataFrame no tipado.

Los Datasets son objetos tipados específicos del dominio que se pueden operar en paralelo utilizando programación funcional o los operadores DSL que ya conoces de la API de DataFrame.

Gracias a esta API única, los desarrolladores de Java ya no corren el riesgo de quedarse atrás. Por ejemplo, cualquier cambio futuro en la interfaz o comportamiento de `groupBy()`, `flatMap()`, `map()`, o `filter()` API será la misma para Java también, ya que es una interfaz única que es común a ambas implementaciones.

### Creando dataset

Spark tiene tipos de datos internos, como `StringType`, `BinaryType`, `IntegerType`, `BooleanType` y `MapType`, que utiliza para mapear sin problemas a los tipos de datos específicos del lenguaje en Scala y Java durante las operaciones de Spark. Este mapeo se realiza a través de encoders, que discutiremos más adelante en este capítulo.

Una forma simple y dinámica de crear un Dataset de muestra es utilizando una instancia de `SparkSession`. En este escenario, con fines ilustrativos, creamos dinámicamente un objeto Scala con tres campos: `uid` (identificador único para un usuario), `uname` (cadena de nombre de usuario generada aleatoriamente) y `usage` (minutos de uso del servidor o servicio):

```
import scala.util.Random._
import spark.implicits._

case class Usage(uid:Int, uname:String, usage: Int)
val r = new scala.util.Random(42)
// Create 1000 instances of scala Usage class
// This generates data on the fly
val data = for (i <- 0 to 1000)
yield (Usage(i, "user-" + r.alphanumeric.take(5).mkString(""),
```

```
r.nextInt(1000)))  
  
// Create a Dataset of Usage typed data  
val dsUsage = spark.createDataset(data)  
dsUsage.show(10)
```

Ahora que tenemos nuestro Dataset generado, `dsUsage`, realicemos algunas de las transformaciones comunes que hemos hecho en capítulos anteriores.

## OPERACIONES

Recuerda que los Datasets son colecciones fuertemente tipadas de objetos específicos del dominio. Estos objetos pueden transformarse en paralelo mediante operaciones funcionales o relacionales. Ejemplos de estas transformaciones incluyen `map()`, `reduce()`, `filter()`, `select()` y `aggregate()`. Como ejemplos de funciones de orden superior, estos métodos pueden tomar funciones lambda, cierres o funciones como argumentos y devolver los resultados. Por lo tanto, se prestan bien a la programación funcional.

Como ejemplo sencillo, usemos `filter()` para obtener todos los usuarios en nuestro Dataset `dsUsage` cuyo uso supere los 900 minutos. Una forma de hacer esto es utilizar una expresión funcional como argumento para el método `filter()`:

## 7. Rdd vs dataframe vs dataset

En Spark, los RDD (Resilient Distributed Dataset), `DataFrame` y `DataSet` son abstracciones que representan conjuntos de datos distribuidos, pero tienen diferencias en términos de funcionalidad y tipo de datos que pueden manejar. Aquí hay una descripción de cada uno:

### **RDD (Resilient Distributed Dataset):**

- Es la abstracción de datos más básica en Spark.
- Colección inmutable y distribuida de objetos que pueden ser procesados en paralelo.
- Puede contener cualquier tipo de objeto.
- Ofrece tolerancia a fallos mediante el seguimiento de la información de cómo construir el conjunto de datos a partir de otros conjuntos de datos.

### **DataFrame:**

- Introducido en Spark 1.3 como una abstracción más rica y eficiente que los RDD.
- Representa una tabla de datos con columnas etiquetadas.
- Tiene un esquema que describe el tipo de datos en cada columna.
- Similar a un `DataFrame` en R o `pandas` en Python.
- Ventajas:
  - Optimización de consultas mediante el Catalyst Optimizer.
  - Puede ser convertido a y desde RDDs.

- Mayor rendimiento en comparación con RDD debido a la optimización de Catalyst y Tungsten.
- Uso:
  - Ideal para el procesamiento de datos estructurados y semiestructurados.
  - Se utiliza comúnmente con Spark SQL para ejecutar consultas SQL en datos distribuidos.

## **DataSet:**

- Introducido en Spark 1.6 como una interfaz orientada a objetos más fuerte que los DataFrames.
- Combina las características de los RDD y los DataFrames.
- Ofrece un sistema de tipo fuerte y expresiones lambda para manipular datos de manera más flexible.
- Ventajas:
  - Beneficios de la inferencia de tipos y rendimiento de los DataFrames, pero con una interfaz de programación más rica y orientada a objetos.
- Uso:
  - Se utiliza cuando es necesario un fuerte sistema de tipos y operaciones más complejas que no son posibles con DataFrames.

En resumen, RDD es la abstracción más básica, DataFrame proporciona un rendimiento optimizado para consultas SQL y manipulación de datos estructurados, mientras que DataSet combina la orientación a objetos con el rendimiento optimizado de los DataFrames. La elección entre ellos depende de los requisitos específicos de tu aplicación y del tipo de operaciones que necesitas realizar en tus datos distribuidos.

## **Resumen**

- Si deseas indicarle a Spark qué hacer, no cómo hacerlo, utiliza DataFrames o Datasets.
- Si buscas semántica rica, abstracciones de alto nivel y operadores DSL, utiliza DataFrames o Datasets.
- Si requieres una seguridad estricta en tiempo de compilación y no te importa crear múltiples clases de casos para un Dataset[T] específico, utiliza Datasets.
- Si tu procesamiento demanda expresiones de alto nivel, filtros, mapas, agregaciones, cálculos de promedios o sumas, consultas SQL, acceso columnar o el uso de operadores relacionales en datos semi-estructurados, utiliza DataFrames o Datasets.
- Si tu procesamiento dicta transformaciones relacionales similares a consultas tipo SQL, utiliza DataFrames.
- Si deseas aprovechar y beneficiarte de la eficiente serialización de Tungsten con Encoders, utiliza Datasets.
- Si buscas unificación, optimización de código y simplificación de APIs en componentes Spark, utiliza DataFrames.

## 8. Spark sql

A nivel programático, Spark SQL permite a los desarrolladores ejecutar consultas compatibles con ANSI SQL:2003 en datos estructurados con un esquema. Desde su introducción en Spark 1.3, Spark SQL ha evolucionado en un motor sustancial sobre el cual se han construido muchas funcionalidades estructuradas de alto nivel. Además de permitirte ejecutar consultas similares a SQL en tus datos, el motor Spark SQL:

- Unifica los componentes de Spark y permite la abstracción a DataFrames/Datasets en Java, Scala, Python y R, lo que simplifica el trabajo con conjuntos de datos estructurados.
- Se conecta al metastore y tablas de Apache Hive.
- Lee y escribe datos estructurados con un esquema específico desde formatos de archivo estructurados (JSON, CSV, Texto, Avro, Parquet, ORC, etc.) y convierte los datos en tablas temporales.
- Ofrece una interfaz interactiva de Spark SQL para una rápida exploración de datos.
- Proporciona un puente hacia (y desde) herramientas externas a través de conectores JDBC/ODBC estándar de bases de datos.
- Genera planes de consulta optimizados y código compacto para la JVM, para su ejecución final.

Además,

- Proporciona el motor sobre el cual se construyen las API Estructuradas de alto nivel que exploramos en el Capítulo 3.
- Puede leer y escribir datos en una variedad de formatos estructurados (por ejemplo, JSON, tablas Hive, Parquet, Avro, ORC, CSV).
- Te permite consultar datos utilizando conectores JDBC/ODBC desde fuentes de datos de inteligencia empresarial externas como Tableau, Power BI, Talend, o desde sistemas de gestión de bases de datos relacionales como MySQL y PostgreSQL.
- Ofrece una interfaz programática para interactuar con datos estructurados almacenados como tablas o vistas en una base de datos desde una aplicación Spark.
- Proporciona una shell interactiva para emitir consultas SQL en tus datos estructurados.
- Admite comandos compatibles con ANSI SQL:2003 y HiveQL.

El `SparkSession`, introducido en Spark 2.0, proporciona un punto de entrada unificado para programar Spark con las API Estructuradas. Puedes utilizar un `SparkSession` para acceder a la funcionalidad de Spark: simplemente importa la clase y crea una instancia en tu código.

Para emitir cualquier consulta SQL, utiliza el método `sql()` en la instancia de `SparkSession`, llamada `spark`, como por ejemplo `spark.sql("SELECT * FROM miNombreDeTabla")`. Todas las consultas `spark.sql` ejecutadas de esta manera devuelven un `DataFrame` en el cual puedes realizar más operaciones de Spark si lo deseas

Vamos a realizar un ejemplo con el dataframe que hemos utilizado en ejemplos anteriores en el que hemos creado un dataframe con las llamadas a los bomberos:

```
import org.apache.spark.sql.{SparkSession, Row}
import org.apache.spark.sql.types.{StructType, StructField, StringType, IntegerType, BooleanType, FloatType}

val fireSchema = StructType(Array(
  StructField("CallNumber", IntegerType, true), StructField("UnitID", StringType, true),
  StructField("IncidentNumber", IntegerType, true), StructField("CallType", StringType, true),
  StructField("CallDate", StringType, true), StructField("WatchDate", StringType, true),
  StructField("CallFinalDisposition", StringType, true), StructField("AvailableDtTm", StringType, true),
  StructField("Address", StringType, true), StructField("City", StringType, true),
  StructField("Zipcode", IntegerType, true), StructField("Battalion", StringType, true),
  StructField("StationArea", StringType, true), StructField("Box", StringType, true),
  StructField("OriginalPriority", StringType, true), StructField("Priority", StringType, true),
  StructField("FinalPriority", IntegerType, true), StructField("ALSUnit", BooleanType, true),
  StructField("CallTypeGroup", StringType, true), StructField("NumAlarms", IntegerType, true),
  StructField("UnitType", StringType, true), StructField("UnitSequenceInCallDispatch", IntegerType,
true), StructField("FirePreventionDistrict", StringType, true), StructField("SupervisorDistrict", StringType,
true), StructField("Neighborhood", StringType, true), StructField("Location", StringType,
true), StructField("RowID", StringType, true), StructField("Delay", FloatType, true) ))

val sfFireFile="/tmp/sf-fire-calls.csv"
val fireDF = spark.read.schema(fireSchema).option("header", "true").csv(sfFireFile)
```

Si queremos realizar consultas sql simplemente tendremos que crear una tabla temporal del dataframe y aplicar el comando `sql` para ejecutar las consultas que deseamos mostrar, estas nos devolverán un dataframe con el que podremos seguir operando:

```
fireDF.createOrReplaceTempView("fireCalls")
val result = spark.sql("SELECT COUNT(*) FROM fireCalls WHERE City IS NULL")
result.show()

+-----+
|count(1)|
+-----+
|      207|
+-----+

// Filtrar filas donde "City" no sea igual a "SF"
spark.sql("SELECT * FROM fireCalls WHERE City <> 'SF'").show()
```

**// Mostrar valores únicos en la columna "City"**

```
spark.sql("SELECT DISTINCT City FROM fireCalls").show()
```

**// Eliminar duplicados en la columna "City" y mostrar los resultados**

```
spark.sql("SELECT DISTINCT City FROM fireCalls").show()
```

**// Ordenar el DataFrame completo por "City" en orden ascendente**

```
spark.sql("SELECT * FROM fireCalls ORDER BY City").show(5)
```

**// Seleccionar el valor mínimo de "CallDate" y el valor máximo de "Priority"**

```
spark.sql("SELECT MIN(CallDate), MAX(Priority) FROM fireCalls").show()
```

**// Seleccionar la suma de "NumAlarms"**

```
spark.sql("SELECT SUM(NumAlarms) FROM fireCalls").show()
```

**// Seleccionar el promedio de "NumAlarms"**

```
spark.sql("SELECT AVG(NumAlarms) FROM fireCalls").show()
```

**// Agrupar por "City" y contar las filas en cada grupo**

```
spark.sql("SELECT City, COUNT(*) FROM fireCalls GROUP BY City").show()
```

**// Agrupar por "City" y calcular la suma de "NumAlarms" como "Total" y el recuento de "Location" como "CountLoc"**

```
spark.sql("SELECT City, SUM(NumAlarms) AS Total, COUNT(Location) AS CountLoc FROM fireCalls GROUP BY City").show()
```

## Spark sql en zeppelin

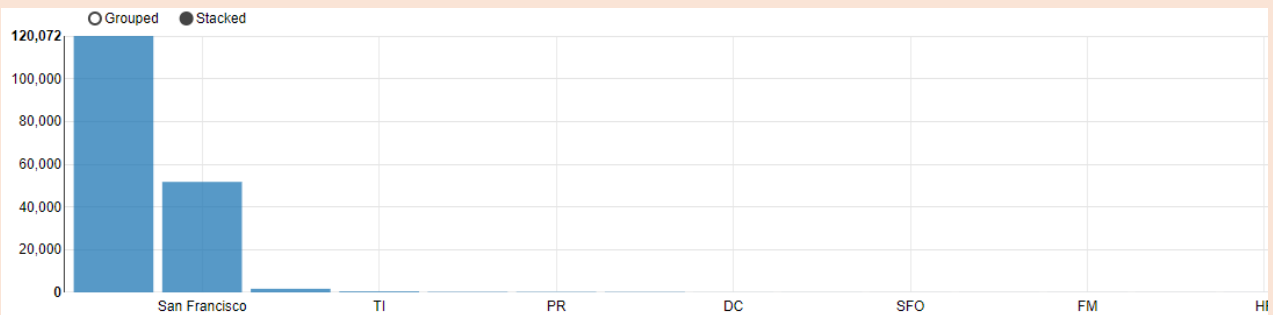
El componente zeppelin ofrece la posibilidad de escribir sentencias SQL importando la ddl `%spark2.sql.`

De esta forma, podemos escribir sql directamente y mostrarlo en una tabla interactiva.

Por ejemplo:

```
%spark2.sql
SELECT City, COUNT(*) FROM fireCalls
where City is not null
GROUP BY City
order by count(*)
```

| City            | count(1) |
|-----------------|----------|
| DC              | 41       |
| TI              | 486      |
| TREASURE ISLAND | 9        |
| San Francisco   | 51739    |
| null            | 207      |
| HP              | 31       |
| YB              | 34       |
| BN              | 9        |



## 8.1. Ejercicios

Vamos a realizar los ejercicios anteriores pero esta utilizaremos Spark SQL.

1. A partir del archivo VentasNulos.csv:
  - c. Elimina las filas que tengan al menos 4 nulos.
  - d. Crea una tabla temporal
  - e. Con las filas restantes crea una vista temporal y obten:
    - i. La media de las ventas de los compañeros (redondeado a entero).
    - ii. El nombre y valor del compañero que menos € ha ganado. (tras agrupar, puedes usar la función min)
2. A partir del archivo movies.tsv, crea un esquema de forma declarativa con los campos:
  - Interprete de tipo string
  - pelicula de tipo string
  - anyo de tipo int

Cada fila del fichero implica que el actor/actriz ha trabajado en dicha película en el año indicado.

Una vez creado el esquema, carga los datos en un DataFrame.

A continuación, mediante el DataFrame API:

- Muestra las películas en las que ha trabajado Murphy, Eddie (I).
- Muestra los intérpretes que aparecen tanto en Superman como en Superman II.

3. Sobre las películas de la sesión anterior:

- g. ¿Cuántas películas diferentes hay?
- h. ¿En cuantas películas ha trabajado Murphy, Eddie (I)?
- i. ¿Cuáles son los actores que han aparecido en más de 30 películas?
- j. ¿En qué película anterior a 1980 aparecen al menos 25 intérpretes?
- k. Muestra la cantidad de películas producidas cada año (solo debe mostrar el año y la cantidad), ordenando el listado por la cantidad de forma descendente.
- l. A partir de la consulta anterior, crea un gráfico de barras que muestre el año y la cantidad de películas, ordenados por fecha.

4. Entre los autores de este libro hay una científica de datos que ama hornear galletas con M&Ms, y premia a sus estudiantes en los estados de EE. UU. donde imparte con frecuencia cursos de aprendizaje automático y ciencia de datos con lotes de esas galletas. Pero ella sigue un enfoque basado en datos, obviamente, y quiere asegurarse de que obtenga los colores correctos de M&Ms en las galletas para los estudiantes en los diferentes estados.

Escribamos un programa de Spark sql que lea un archivo con más de 100,000 entradas (donde cada fila o línea tiene un <estado, color\_mnm, recuento>) y calcule y agregue los recuentos para cada color y estado. Estos recuentos agregados nos dicen los colores de M&Ms preferidos por los estudiantes en cada estado. El listado completo se encuentra en el csv "mnm\_dataset.csv".

## 9. Carga y almacenamiento distintos tipos archivos

Con las instrucciones read y write de de spark se pueden hacer importaciones y exportaciones en distintos tipos de archivos.

A continuación vamos a ver las más comunes

### 9.1. CSV

Utilizando **spark.read.csv("path")** o **spark.read.format("csv").load("path")** se puede leer uno o varios archivos csv a un dataframe.



Por ejemplo:

```
val df = spark.read.csv("src/main/resources/zipcodes.csv")  
df.printSchema()
```

Si el archivo tiene un encabezado de columnas hay que especificarlo en la sección options forma:

```
val df = spark.read.option("header",true).csv("src/main/resources/zipcodes.csv")
```

Además, se pueden leer varios archivos csv especificando las rutas separadas por comas o leer todos los archivos de un directorio especificando la ruta raíz.

```
val df = spark.read.csv("path1,path2,path3")  
val df = spark.read.csv("Folder path")
```

La función de importación de csv tiene varias opciones al leer el archivo.

#### *Delimiter*

Se utiliza para especificar el delimitador de columna del archivo CSV. De forma predeterminada, es el carácter de coma (,), pero se puede configurar como barra vertical (|), tabulación, espacio o cualquier carácter usando esta opción.

```
val df2 = spark.read.options(Map("delimiter"->"|")).csv("src/main/resources/zipcodes.csv")
```

#### *inferSchema*

El valor predeterminado es false. Cuando esta a true significa que se infieren automáticamente todos los tipos de datos.

```
val df2 = spark.read.options(Map("inferSchema"->"true", "delimiter"->"|")).csv("src/main/resources/zipcodes.csv")
```

Además de estas opciones hay muchas [más opciones](#).

Utilizando **spark.write** se puede escribir un dataframe en un archivo.

```
df2.write.option("header", "true").csv("/tmp/spark_output/zipcodes")
```

Mientras se escribe un archivo CSV, se pueden utilizar varias opciones. Estas opciones están disponibles en la [documentación](#)

## 9.2. JSON

Al igual que con el csv se utilizan las sentencias **spark.read.json("path")** leer un archivo JSON de una sola línea y de varias líneas (múltiples líneas) a un dataframe y **dataframe.write.json("path")** guardar o escribir en un archivo JSON.

Al igual que con el csv existen múltiples opciones que puedes consultar en la [documentación](#) de spark.

### 9.3. PARQUET

Apache Parquet es un formato de archivo en columnas que proporciona optimizaciones para acelerar las consultas y es un formato de archivo mucho más eficiente que CSV o JSON, compatible con muchos sistemas de procesamiento de datos.

Es compatible con la mayoría de los marcos de procesamiento de datos en los sistemas de eco Hadoop . Proporciona esquemas eficientes de compresión y codificación de datos con rendimiento mejorado para manejar datos complejos de forma masiva.

Spark SQL brinda soporte para leer y escribir archivos Parquet que capturan automáticamente el esquema de los datos originales. También reduce el almacenamiento de datos en un 75% en promedio. A continuación se detallan algunas ventajas de almacenar datos en formato parquet. Spark admite de forma predeterminada Parquet en su biblioteca, por lo que no necesitamos agregar ninguna biblioteca de dependencia.

De forma similar a los csv y json podemos leer y escribir archivos parquet .

```
val parqDF = spark.read.parquet("/tmp/output/people.parquet")
```

Spark proporciona la capacidad de agregar registros a archivos parquet existentes de la siguiente forma

```
df.write.mode('append').parquet("/tmp/output/people.parquet")
```

Para más información consulta la [documentación](#) oficial.

## 10. Spark streaming

Cuando el procesamiento se realiza en streaming:

- Los datos se generan de manera continuada desde una o más fuentes de datos.
- Las fuentes de datos, por lo general, envían los datos de forma simultánea.
- Los datos se reciben en pequeños fragmentos (del orden de KB).

Vamos a considerar un stream como un flujo de datos continuo e ilimitado, sin un final definido que aporta datos a nuestros sistemas cada segundo.

El desarrollo de aplicaciones que trabajan con datos en streaming suponen un mayor reto que las aplicaciones batch, dada la impredecibilidad de los datos, tanto su ritmo de llegada como su orden.

Uno de los casos de uso más comunes del procesamiento en streaming es realizar algún cálculo agregado sobre los datos que llegan y resumirlos/sintetizarlos en un destino externo para que luego ya sea una aplicación web o un motor de analítica de datos los consuman.

Las principales herramientas para el tratamiento de datos en streaming son **Apache Samza**, **Apache Flink**, **Apache Kafka** (de manera conjunta con Kafka Streams) y por supuesto, **Apache Spark**.

## 10.1. Streaming en spark

Spark Streaming es una extensión del núcleo de Spark que permite el procesamiento de flujos de datos en vivo ofreciendo tolerancia a fallos, un alto rendimiento y altamente escalable.

Los datos se pueden ingestar desde diversas fuentes de datos, como Kafka, sockets TCP, etc.. y se pueden procesar mediante funciones de alto nivel, ya sea mediante el uso de RDD y algoritmos MapReduce, o utilizando DataFrames y la sintaxis SQL. Finalmente, los datos procesados se almacenan en sistemas de ficheros, bases de datos o cuadros de mandos.



De hecho, podemos utilizar tanto Spark MLlib y sus algoritmos de machine learning como el procesamiento de grafos en los flujos de datos.

Spark dispone dos soluciones para trabajar con datos en streaming:

Spark DStream: más antigua, conocida como la primera generación, basada en RDDs

Spark Structured Streaming basada en el uso de DataFrames y diseñada para construir aplicaciones que puedan reaccionar a los datos en tiempo real.

Vamos a trabajar únicamente Spark Structured Streaming.

## 10.2. Structured streaming

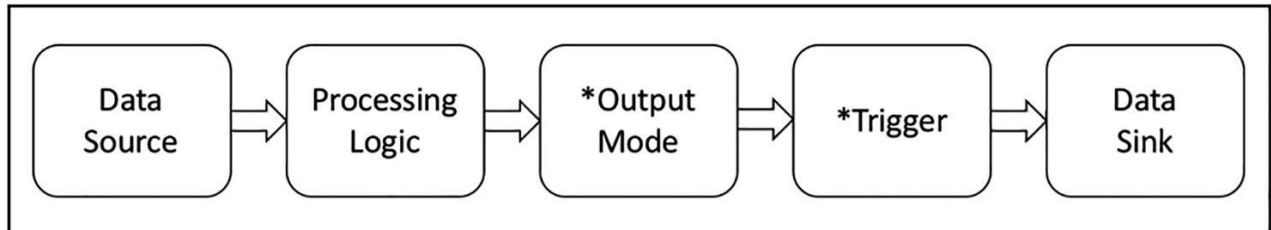
Spark Structured Streaming es la segunda generación de motor para el tratamiento de datos en streaming, y fue diseñado para ser más rápido, escalable y con mayor tolerancia a los errores que DStream, ya que utiliza el motor de Spark SQL.

Además, podemos expresar los procesos en streaming de la misma manera que realizaríamos un proceso batch con datos estáticos. El motor de Spark SQL se encarga de ejecutar los datos de forma continua e incremental, y actualizar el resultado final como datos streaming. Para ello, podemos utilizar el API de Java, Scala, Python o R para expresar las agregaciones, ventanas de eventos, joins de stream a batch, etc....

Finalmente, el sistema asegura la tolerancia de fallos mediante la entrega de cada mensaje una sola vez (exactly-once) a través de checkpoints y logs.

Los pasos esenciales al codificar una aplicación en streaming son:

- Especificar uno o más fuentes de datos
- Desarrollar la lógica para manipular los flujos de entrada de datos mediante transformaciones de DataFrames,
- Definir el modo de salida
- Definir el trigger que provoca la lectura
- Indicar el destino de los datos (data sink) donde escribir los resultados.



Debido a que tanto el modo de salida como el trigger tienen valores por defecto, es posible que no tengamos que indicarlos ni configurarlos, lo que reduce el desarrollo de procesos a un bucle infinito de leer, transformar y enviar al destino (read + transform + sink). Cada una de las iteraciones de ese bucle infinito se conoce como un micro-batch, las cuales tienen unas latencias situadas alrededor de los 100 ms.

### 10.3. Ejemplo spark streaming

Para ver nuestro primer caso de uso, vamos a realizar un proceso de contar palabras sobre un flujo continuo de datos que proviene de un socket.

Para ello, en un terminal, abrimos un listener de Netcat en el puerto 9999:

```
nc -lk 9999
```

Tras arrancar Netcat, ya podemos crear nuestra aplicación Spark (vamos a indicar que cree 2 hilos, lo cual es el mínimo necesario para realizar streaming, uno para recibir y otro para procesar), en la cual tenemos diferenciadas:

- la fuente de datos: creación del flujo de lectura mediante **readStream** que devuelve un **DataStreamReader** que utilizaremos para cargar un **DataFrame**.
- la lógica de procesamiento ya sea mediante *DataFrames API* o *Spark SQL*.
- la persistencia de los datos mediante **writeStream** que devuelve un **DataStreamWriter** donde indicamos el modo de salida, el cual, al iniciarlo con **start** nos devuelve un **StreamingQuery**
- y finalmente el cierre del flujo de datos a partir de la consulta en streaming mediante **awaitTermination**.

```
import org.apache.spark.sql.SparkSession

val spark = SparkSession.builder.appName("Streaming
WordCount").config("spark.streaming.stopGracefullyOnShutdown", "true").getOrCreate()
```

```
import org.apache.spark.sql.functions._
import org.apache.spark.sql.streaming.StreamingQuery
import org.apache.spark.sql.streaming.StreamingQueryException

val lineasDF = spark.readStream.format("socket").option("host", "localhost").option("port", "9999").load()

// Leemos las líneas y las pasamos a palabras.
// Sobre ellas, realizamos la agrupación count (transformación)
val palabrasDF = lineasDF.select(explode(split(col("value"), " ")).alias("palabra"))
val cantidadDF = palabrasDF.groupBy("palabra").count()

// Mostramos las palabras por consola (sink)
// En Spark Streaming, la persistencia se realiza mediante writeStream
// y en vez de realizar un save, ahora utilizamos start
val wordCountQuery: StreamingQuery = cantidadDF.writeStream.format("console").outputMode("complete")
.start()

// dejamos Spark a la escucha
try {
  wordCountQuery.awaitTermination()
} catch {
  case e: StreamingQueryException =>
    e.printStackTrace()
}
```

Conforme escribamos en el terminal de Netcat irán apareciendo en la consola de Spark los resultados:

```
sandbox-hdp login: root
root@sandbox-hdp.hortonworks.com's
Last login: Tue Nov 28 09:31:04 20
[root@sandbox-hdp ~]# nc -lk 9999
hola
esta
es una prueba
para comprobar si funciona
spark streaming
```

```
import org.apache.spark.sql.functions._
import org.apache.spark.sql.streaming.StreamingQuery
import org.apache.spark.sql.streaming.StreamingQueryException

val lineasDF = spark.readStream.format("socket").option("host", "localhost").option("port", "9999").load()

// Leemos las líneas y las pasamos a palabras.
// Sobre ellas, realizamos la agrupación count (transformación)
val palabrasDF = lineasDF.select(explode(split(col("value"), " ")).alias("palabra"))
val cantidadDF = palabrasDF.groupBy("palabra").count()

// Mostramos las palabras por consola (sink)
// En Spark Streaming, la persistencia se realiza mediante writeStream
// y en vez de realizar un save, ahora utilizamos start
val wordCountQuery: StreamingQuery = cantidadDF.writeStream.format("console").outputMode("complete").start()

// dejamos Spark a la escucha
try {
  wordCountQuery.awaitTermination()
} catch {
  case e: StreamingQueryException =>
    e.printStackTrace()
}
```

```
+-----+-----+
| palabra|count|
+-----+-----+
para	1
si	1
	3
comprobr	1
funciona	1
spark	1
esta	1
streaming	1
una	1
hola	1
es	1
prueba	1
+-----+-----+
```

Al ejecutar la consulta, Spark crea un proceso a la escucha de manera ininterrumpida de nuevos datos. Mientras no lleguen datos, Spark queda a la espera, de manera que cuando llegue algún dato al flujo de entrada, se creará un nuevo micro-batch, lo que lanzará un nuevo job de Spark.

Si queremos detenerlo, podemos hacerlo de forma explícita:

```
wordCountQuery.stop()
```

Una buena práctica es configurar la SparkSession mediante la propiedad `spark.streaming.stopGracefullyOnShutdown` para que detenga el streaming al finalizar el proceso:

```
spark = SparkSession.builder.appName("Streaming WordCount")
  .config("spark.streaming.stopGracefullyOnShutdown", "true").getOrCreate()
```

Por defecto, Spark utiliza 200 particiones para barajar los datos. Como no tenemos muchos datos, para obtener un mejor rendimiento, podemos reducir su cantidad mediante la propiedad `spark.sql.shuffle.partitions`:

```
spark = SparkSession.builder.appName("Streaming WordCount")
  .config("spark.streaming.stopGracefullyOnShutdown", "true").config("spark.sql.shuffle.partitions", 3)
  .getOrCreate()
```

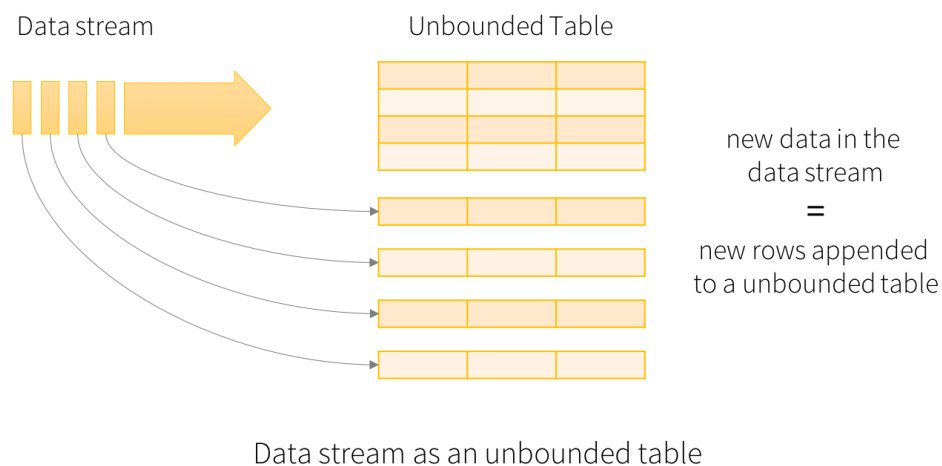
Es recomendable indicar el nombre de la consulta mediante el método `queryName`, el cual nos sirve luego para monitorizar la ejecución del flujo:

```
wordCountQuery = cantidadDF.writeStream.queryName("Caso1 WordCount").format("console")
```

```
.outputMode("complete").start()
```

## 10.4. Elementos


La idea básica al trabajar los datos en streaming es similar a tener una tabla de entrada de tamaño ilimitado, y conforme llegan nuevos datos, tratarlos como un nuevo conjunto de filas que se adjuntan a la tabla.



### Fuentes de datos

Mientras que en el procesamiento batch las fuentes de datos son datasets estáticos que residen en un almacenamiento como pueda ser un sistema local, HDFS o S3, al hablar de procesamiento en streaming las fuentes de datos generan los datos de forma continuada, por lo que necesitamos otro tipo de fuentes.

Structured Streaming ofrece un conjunto predefinido de fuentes de datos que se leen a partir de un `DataStreamReader`. Los tipos existentes son:

-  **Fichero:** permite leer ficheros desde un directorio como un flujo de datos, con soporte para ficheros de texto, CSV, JSON, Parquet, ORC, etc...

```
# Lee todos los ficheros csv de un directorio
val esquemaUsuario = new StructType().add("nombre", StringType).add("edad", IntegerType)
// Lee todos los archivos CSV en un directorio
val csvDF = spark.readStream.option("sep", ";").schema(esquemaUsuario).csv("/path/al/directorio")
```

Podemos configurar otras opciones como `maxFilesPerTrigger` con la cantidad de archivos a cargar en cada trigger, así como la política de lectura cuando su número sea mayor de uno mediante la propiedad booleana `latestFirst`.

- ✚ **Kafka:** para leer datos desde brokers Kafka (versiones 0.10 o superiores). Realizaremos un par de ejemplos en los siguientes apartados.

```
val kafkaDF = spark.readStream.format("kafka").option("kafka.bootstrap.servers",  
"host1:port1,host2:port2").option("subscribe", "iabd-topic").load()
```

- ✚ **Socket:** lee texto UTF8 desde una conexión socket (es el que hemos utilizado en el caso de uso 1). Sólo se debe utilizar para pruebas ya que no ofrece garantía de tolerancia de fallos de punto a punto.

```
val socketDF = spark.readStream.format("socket").option("host", "localhost").option("port",  
9999).load()
```

- ✚ **Rate:** Genera datos indicando una cantidad de filas por segundo, donde cada fila contiene un timestamp y el valor de un contador secuencial (la primera fila contiene el 0). Esta fuente también se utiliza para la realización de pruebas y benchmarking.

```
val socketDF = spark.readStream.format("rate").option("rowsPerSecond", 1).load()
```

- ✚ **Tabla (desde Spark 3.1):** Carga los datos desde una tabla temporal de SparkSQL, la cual podemos utilizar tanto para cargar como para persistir los cálculos realizados. Más información en la documentación oficial.

```
val tablaDF = spark.readStream.table("clientes")
```

## Sink

De la misma manera, también tenemos un conjunto de *Sinks* predefinidos como destino de los datos, que se escriben a partir de un **DataStreamWriter** mediante el interfaz **writeStream**:

- ✚ **Fichero:** Podemos almacenar los resultados en un sistema de archivos, HDFS o S3, con soporte para los formatos CSV, JSON, ORC y Parquet.

```
// Otros valores pueden ser "json", "csv", etc...  
df.writeStream.format("parquet").option("path", "/path/al/directorio").start()
```

- ✚ **Kafka:** Envía los datos a un clúster de *Kafka*:

```
val query: StreamingQuery = df.writeStream.format("kafka").option("kafka.bootstrap.servers",  
"host1:port1,host2:port2").option("topic", "miTopic").start()
```

- ✚ **Foreach y ForeachBatch:** permiten realizar operaciones y escribir lógica sobre la salida de una consulta de *streaming*, ya sea a nivel de fila (*foreach*) como a nivel de micro-batch (*foreachBatch*). Más información en la documentación oficial.
- ✚ **Consola:** se emplea para pruebas y depuración y permite mostrar el resultado por consola.

```
val query: StreamingQuery = df.writeStream.format("console").start()
```



Admite las opciones `numRows` para indicar las filas a mostrar y `truncate` para truncar los datos si las filas son muy largas.

- ✚ **Memoria:** se emplea para pruebas y depuración, ya que sólo permite un volumen pequeño de datos para evitar un problema de falta de memoria en el driver para almacenar la salida. Los datos se almacenan en una tabla temporal a la cual podemos acceder desde *SparkSQL*:

```
val query: StreamingQuery = df.writeStream.format("memory").queryName("nombreTabla").start()
```

## Modos de salida

El modo de salida determina cómo salen los datos a un sumidero de datos. Existen tres opciones:

- ✚ **Añadir (append):** para insertar los datos, cuando sabemos que no vamos a modificar ninguna salida anterior, y que cada batch únicamente escribirá nuevos registros. Es el modo por defecto.
- ✚ **Modificar (update):** similar a un upsert, donde veremos solo registros que, bien son nuevos, bien son valores antiguos que debemos modificar.
- ✚ **Completa (complete):** para sobrescribir completamente el resultado, de manera que siempre recibimos la salida completa.

## Transformaciones

Dentro de Spark Structured Streaming tenemos dos tipos de transformaciones:

- ✚ **Sin estado (stateless):** los datos de cada micro-batch son independientes de los anteriores, y por tanto, podemos realizar las transformaciones `select`, `filter`, `map`, `flatMap`, `explode`. Es importante destacar que estas transformaciones no soportan el modo de salida `complete`, por lo que sólo podemos utilizar los modos `append` o `update`.
- ✚ **Con estado (stateful):** aquellas que implica realizar agrupaciones, agregaciones, `windowing` y/o `joins`, ya que mantienen el estado entre los diferentes micro-batches. Hay que destacar que un abuso del estado puede causar problemas de falta de memoria, ya que el estado se almacena en la memoria de los ejecutores (executors). Por ello, Spark ofrece dos tipos de operaciones con estado:
  - **Gestionadas (managed):** Spark gestiona el estado y libera la memoria conforme sea necesario.
  - **Sin gestionar (unmanaged):** permite que el desarrollador defina las políticas de limpieza del estado (y su liberación de memoria), por ejemplo, a partir de políticas basadas en el tiempo. Hoy en día, las transformaciones sin gestionar sólo están disponibles mediante Java o Scala.

Además, hay que tener en cuenta que no todas las operaciones que realizamos con DataFrames están soportadas al trabajar en streaming, como pueden ser `show`, `describe`, `count` (aunque sí que podemos contar sobre agregaciones/funciones ventana), `limit`, `distinct`, `cube` o `sort` (podemos ordenar en algunos casos después de

haber realizado una agregación), ya que los datos no están acotados y provocará una excepción del tipo `AnalysisException`.

## Triggers

Un trigger define el intervalo (timing) temporal de procesamiento de los datos en streaming, indicando si la consulta se ejecutará como un micro-batch mediante un intervalo fijo o con una consulta con procesamiento continuo.

Así pues, un trigger es un mecanismo para que el motor de Spark SQL determine cuando ejecutar la computación en streaming.

Los posibles tipos son:

- ✚ **Sin especificar:** de manera que cada micro-batch se va a ejecutar tan pronto como lleguen datos.
- ✚ **Por intervalo de tiempo:** mediante la propiedad `processingTime`. Si indicamos un intervalo de un minuto, una vez finalizado un job, si no ha pasado un minuto, se esperará a ejecutarse. Si el micro-batch tardase más de un minuto, el siguiente se ejecutaría inmediatamente. Así pues, de esta manera, Spark permite coleccionar datos de entrada y procesarlos de manera conjunta (en vez de procesar individualmente cada registro de entrada).
- ✚ **Un intervalo:** mediante la propiedad `once` de manera que funciona como un proceso batch estándar, creando un único proceso micro-batch, o con la propiedad `availableNow` para leer todos los datos disponibles hasta el momento mediante múltiples batches.
- ✚ **Continuo**, mediante la propiedad `continuous`, para permitir latencias del orden de milisegundos mediante Continuous Processing. Se trata de una opción experimental desde la versión 2.3 de Spark.

Los triggers se configuran al persistir el `DataFrame`, tras indicar el modo de salida mediante el método `trigger`:

```
val wordCountQuery: StreamingQuery = cantidadDF.writeStream.format("console")  
  .outputMode("complete").trigger(processingTime="1 minute").start()
```

## 10.5. Ejercicios

1. En la siguiente [URL](#) tienes el **caso de uso 2: facturas** realizado con python, intenta realizarlo en scala y pruebalos. Adjunta script y resultado.
2. Intenta ejecutar el siguiente ejemplo de [twitter](#) y muéstrame los resultados.

### 3. Bibliografia

<https://www.diegocalvo.es/tutorial-de-scala/>

<https://aitor-medrano.github.io/iabd2223/spark/02dataframeAPI.html#creando-dataframes>

[https://dit.gonzalonazareno.org/gestiona/proyectos/2016-17/Apache\\_spark-alejandro\\_palomino.pdf](https://dit.gonzalonazareno.org/gestiona/proyectos/2016-17/Apache_spark-alejandro_palomino.pdf)

<https://sparkbyexamples.com/>

Jules S. Damji, Brooke Wenig, Tathagata Das, and Denny Lee (2020) **Learning Spark**. O'Reilly Media.