IES SAN ANDRÉS

ESPECIALIZACIÓN EN INTELIGENCIA ARTIFICIAL Y BIG DATA

BIG DATA APLICADO

UT03: PROCESAMIENTO PARALELO EN APACHE HADOOP: SPARK

ÍNDICE

💡 Ejemplos Prácticos Recomendados

Para un curso práctico, es vital que implementen ejemplos de principio a fin.

- **Ejemplo 1:** ETL (Extract, Transform, Load) de grandes volúmenes de datos.
- **Ejemplo 2:** Construcción de un Sistema de Recomendación simple con MLlib.
- **Ejemplo 3:** Análisis de Sentimiento en tiempo real usando Structured Streaming y datos de ejemplo simulados (e.g., de Twitter/Kafka).

1 FUNDAMENTOS DE APACHE SPARK

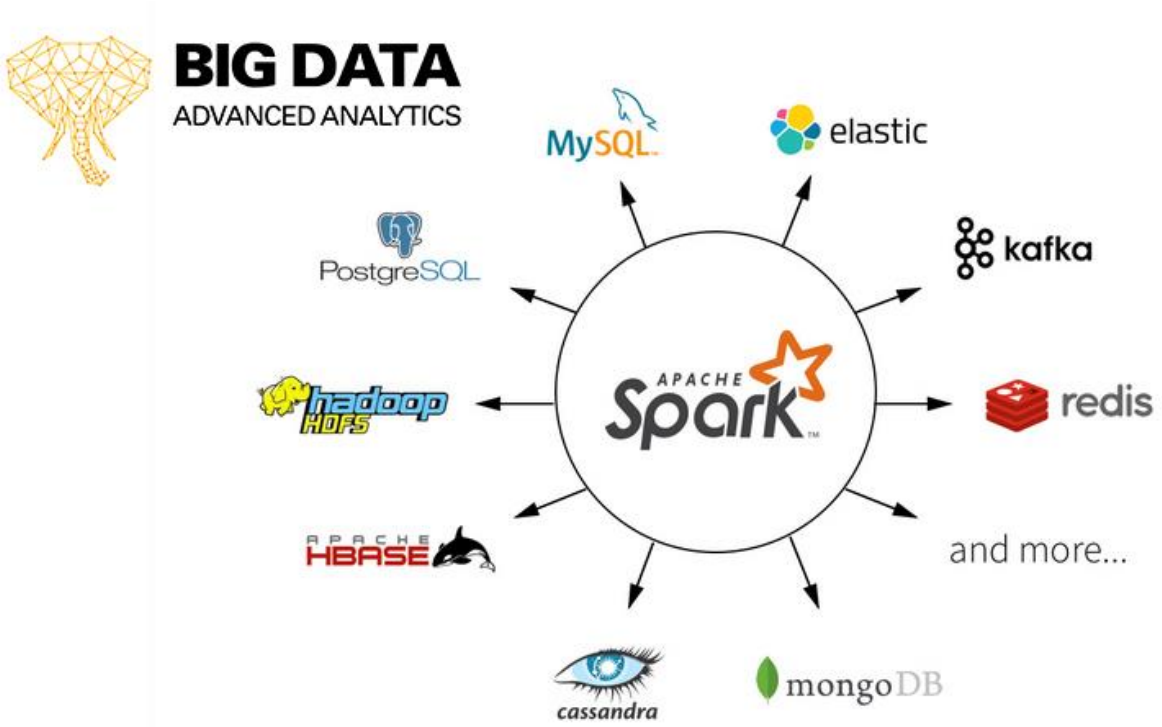
1 FUNDAMENTOS DE APACHE SPARK

1.1 ¿QUÉ ES APACHE SPARK?

Apache Spark es un **motor de análisis y procesamiento de datos distribuidos y de código abierto** diseñado para manejar grandes volúmenes de datos de manera eficiente y rápida

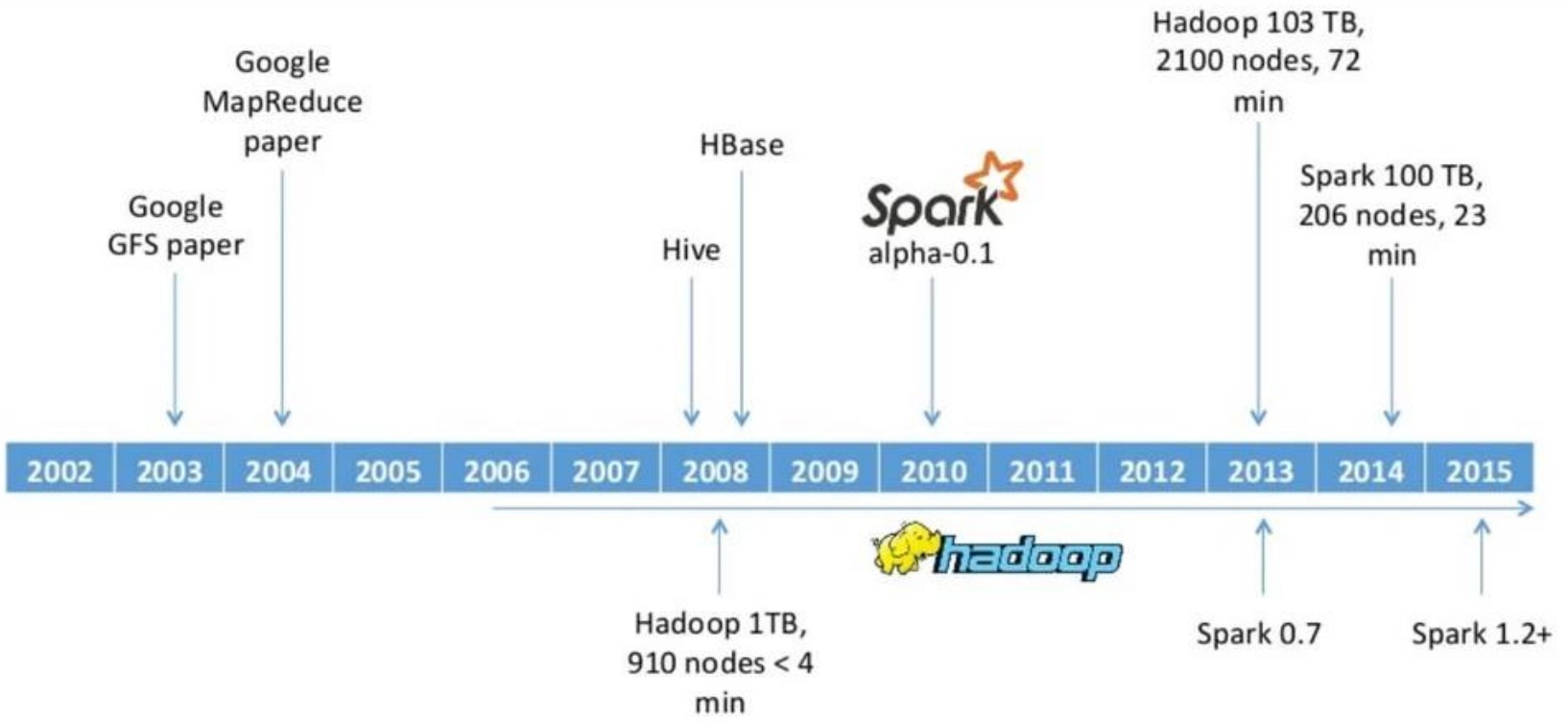
Su principal fortaleza es la capacidad de ejecutar tareas de **procesamiento en memoria**, lo que le hace extremadamente rápido.

Apache Spark puede **leer datos de diversas fuentes** (HDFS, S3, Cassandra, Kafka,..) y aplicar transformaciones complejas en paralelo a través de un clúster de máquinas.



HISTORIA DE APACHE SPARK

- **2009**: nace como proyecto de investigación en el AMPLab de la Universidad de Berkeley, liderado por Matei Zaharia.
- **2010**: el código se publica como Open Source
- **2013**: se dona a la Apache Software Foundation
- **2014**: se convierte en un proyecto de nivel superior de Apache y rápidamente gana popularidad, superando a MapReduce como estándar de facto para el procesamiento de Big Data


Google GFS paper
Google MapReduce paper
Hive
HBase
Spark alpha-0.1
Hadoop 103 TB, 2100 nodes, 72 min
Spark 100 TB, 206 nodes, 23 min
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
hadoop
Hadoop 1TB, 910 nodes < 4 min
Spark 0.7
Spark 1.2+

ARQUITECTURA DE APACHE SPARK

La arquitectura de Spark está diseñada en torno al concepto de procesamiento en un clúster de nodos.

Driver Program (Programa Controlador)

Es el proceso que ejecuta el programa principal (p.e. un script de Python o Scala).

Contiene la **Spark Session** (o el antiguo **Spark Context**), que es el punto de entrada a toda la funcionalidad de Spark.

Analiza, optimiza y distribuye la ejecución del trabajo a los *Executors* en el clúster.

Cluster Manager(Gestor del clúster)

Es un servicio externo (como **YARN**, **Mesos** o el propio **Spark Standalone**) que se encarga de adquirir recursos (cores y memoria) en el clúster.

Lanza los procesos *Executor* en los *Worker Nodes*.

Worker Nodes (Nodos trabajadores)

Las máquinas físicas o virtuales **donde se ejecutan las tareas**.

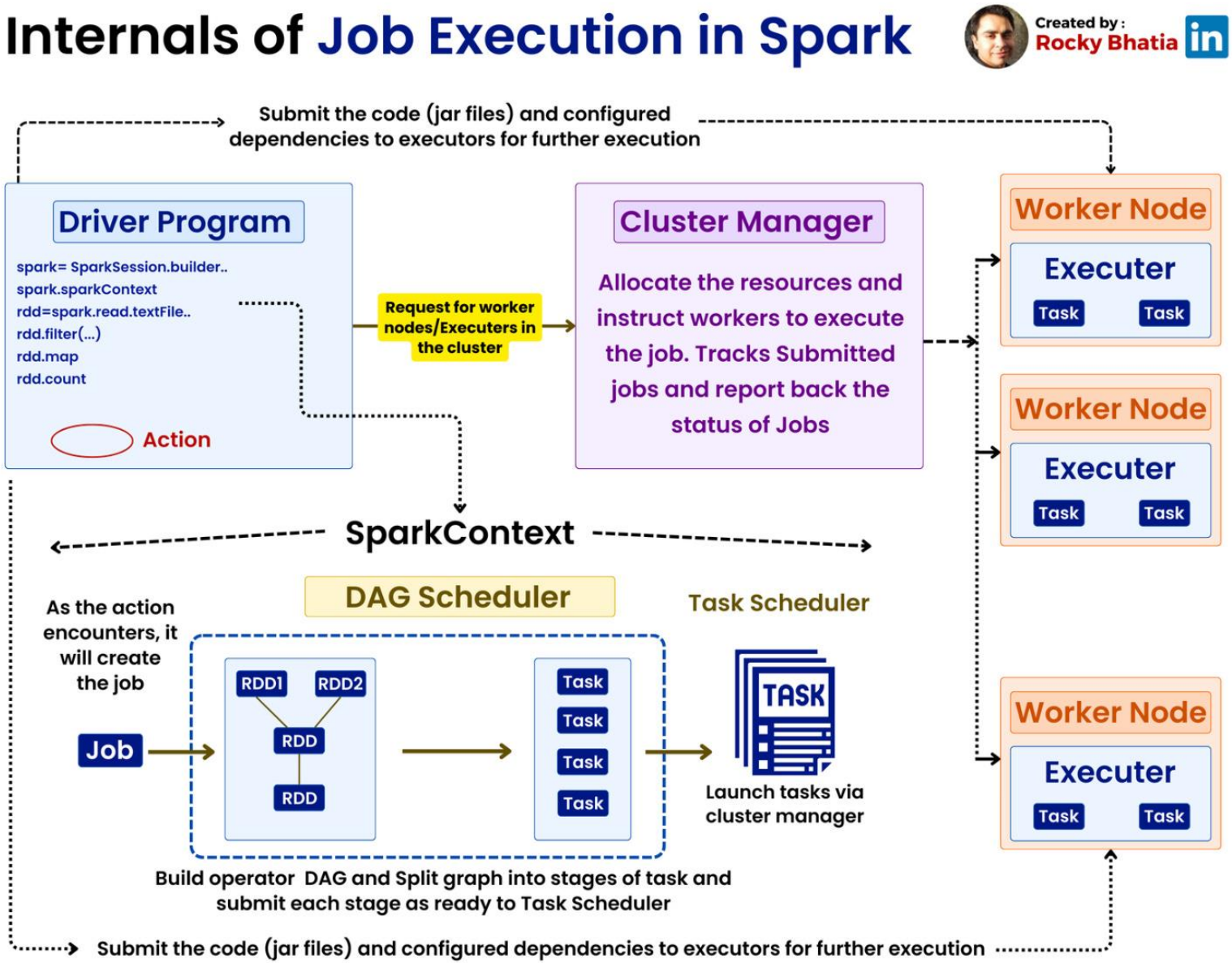
Cada *Worker Node* aloja uno o más procesos *Executor*.

Executors (Ejecutores)

Los **procesos que realmente ejecutan las tareas** (*tasks*) de cómputo en los *Worker Nodes*.

Almacenan los datos en caché de forma persistente.

Reportan el estado y los resultados de vuelta al *Driver Program*.


Internals of Job Execution in Spark
Created by : Rocky Bhatia
Submit the code (jar files) and configured dependencies to executors for further execution
Driver Program
spark= SparkSession.builder..
spark.sparkContext
rdd=spark.read.textFile..
rdd.filter(...)
rdd.map
rdd.count
Action
Request for worker nodes/Executers in the cluster
Cluster Manager
Allocate the resources and instruct workers to execute the job. Tracks Submitted jobs and report back the status of Jobs
Worker Node
Executer
Task
Task
Worker Node
Executer
Task
Task
Worker Node
Executer
Task
Task
SparkContext
DAG Scheduler
Task Scheduler
As the action encounters, it will create the job
Job
RDD1
RDD2
RDD
RDD
Task
Task
Task
Task
TASK
Launch tasks via cluster manager
Build operator DAG and Split graph into stages of task and submit each stage as ready to Task Scheduler
Submit the code (jar files) and configured dependencies to executors for further execution

RDD (Resilient Distributed Dataset)

Es una **colección inmutable, distribuida y tolerante a fallos de objetos** en el clúster de Spark, que se puede operar en paralelo.

Es la **abstracción** central de Spark Core

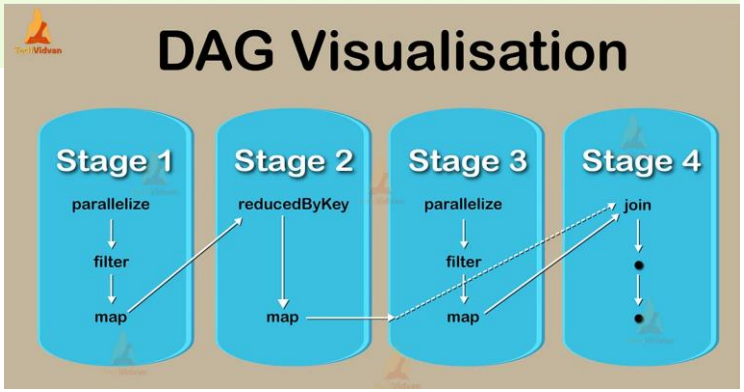
Sus principales características son:

Resiliente (tolerante a fallos)

Un RDD es una **colección de elementos tolerante a fallos que se pueden operar en paralelo**.

Si un nodo falla, Spark puede reconstruir las particiones de datos perdidas desde el origen, sin perder el estado del cálculo.

Para ello utiliza los **Grafos Acíclicos Dirigidos** (DAG)



Distribuido

Un RDD divide automáticamente los datos en varias piezas lógicas llamadas **particiones**.

Cada partición reside en un *worker* diferente del clúster y **se procesa en paralelo**.

El usuario escribe una operación (p.e. map) y Spark se encarga de dividir y orquestar miles de tareas en los diferentes nodos.

Dataset (conjunto de datos)

El *dataset* describe el contenido del RDD

Los RDDs son colecciones de objetos (como una lista o array) que se encuentran dispersos en memoria o disco de las máquinas del clúster.

Los RDDs son **inmutables**. Una vez que se crea un RDD no se puede cambiar. Esto es esencial para la coherencia y la recuperación.

Evaluación perezosa

Las operaciones con RDDs se dividen en dos categorías:

- **Transformaciones**:
  - Crean un nuevo RDD a partir del existente (map, reduce, ..)
  - No ejecutan nada inmediatamente, simplemente registran la operación en el DAG.
- **Acciones**
  - Son operaciones que activan el cómputo (collect, count, save, …)
  - Cuando se llama a una acción, Spark mira el DAG y planifica la ejecución optimizada de todas las transformaciones

Las transformaciones definen la eficiencia del RDD.

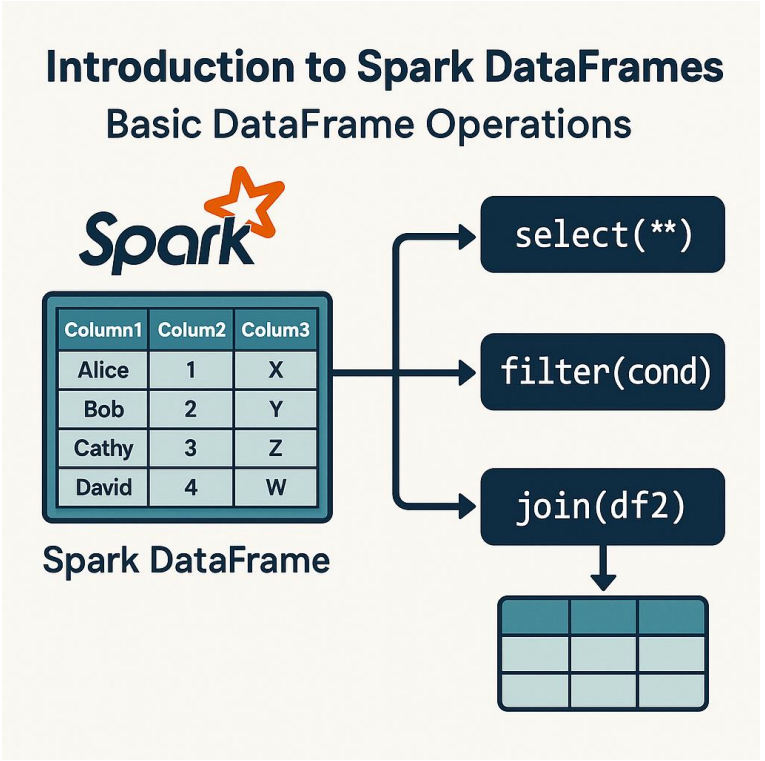
Hay dos tipos:

- **Transformaciones estrechas**:
  - Cada partición del RDD padre contribuye a una única partición en el RDD resultante (p.e. map, filter)
  - Esto es rápido porque no requiere mover datos entre nodos
- **Transformaciones anchas**:
  - Una partición del RDD padre puede contribuir a múltiples particiones en el RDD resultante (p.e. groupByKey, reduceByKey)
  - Esto requiere un **shuffling**, que es la transferencia costosa de datos a través de la red del clúster

Dataframes

Un dataframe es una **colección distribuida de datos organizados en columnas con nombre**

Conceptualmente es equivalente a una tabla de una base de datos relacional o a un *dataframe* de Pandas, pero con **optimizaciones masivas para ejecutarse en un clúster** de computadoras.



Sus principales características son:

- **Estructurados**: los dataframes tienen un esquema definido (frente a los RDDs que son colecciones de objetos arbitrarios)
- **Distribuidos**: los datos no residen en una sola máquina.
- **Inmutables**: una vez creado, un dataframe no se puede modificar, si se quiere cambiar algo hay que crear un nuevo dataframe a través de una transformación.
- **Evaluación perezosa:** al igual que pasa con los RDDs, Spark no ejecuta nada inmediatamente, sino que construye un plan lógico y solo procesa los datos cuando se llama a una acción (show(), count(), write())
- **API unificada**: pueden ser manipulados desde diferentes lenguajes (Scala, Java, Python, R). También permiten ejecutar consultas SQL directamente sobre ellos creando vistas temporales.

2 DATAFRAMES: INGESTA Y TRANSFORMACIÓN

2 DATAFRAMES: INGESTA Y TRANSFORMACIÓN

2.1 INTRODUCCIÓN A PYSPARK

PRIMERA APLICACIÓN EN SPARK

Víctor J. González

Vamos a lanzar Spark en un entorno contenerizado (https://vgonzalez165.github.io/docker_resources/compose/spark/) compuesto por un master, un worker y un tercer contenedor con Jupyter que nos servirá para programar con PySpark

●	spark	-	-	-
●	jupyter-spark-driver	e845bbfd37a2	spark-jupyter-driver	8899:8888
●	worker-1	62d4a6bb9690	spark:3.5.7-scala2.12-java11-pythc	8081:8081
●	master	0ee6e345dcb6	spark:3.5.7-scala2.12-java11-pythc	7077:7077 8080:8080

Conexión a Spark

Lo primero que tenemos que hacer es conectarnos a Spark

```
from pyspark.sql import SparkSession

try:
    spark = ( SparkSession.builder
                .appName("Haciendo pruebas")
                .master("spark://spark-master:7077")
                .getOrCreate()
            )

    print("SparkSession iniciada correctamente.")
except Exception as e:
    print("Error en la conexión")
    print(e)

sc = spark.sparkContext
```

Comenzamos creando una sesión de Spark

Conectamos al contenedor master de Spark a través del puerto 7077

Carga de datos en Spark

En este primer programa simularemos datos de prueba

```
data = [
    ("Ana", 15),
    ("Carlos", 22),
    ("Luis", 10),
    ("Marta", 35)
]
columns = ["nombre", "edad"]

df = spark.createDataFrame(data, columns)
```

Los dataframes tienen **estructura tabular**. En este caso crearemos una tabla de 2 columnas y 4 registros

Las columnas de las tablas tienen que tener un nombre

Creamos el dataframe

```
df.show()
```

```
+------+----+
|nombre|edad|
+------+----+
|   Ana|  15|
|Carlos|  22|
|  Luis|  10|
| Marta|  35|
+------+----+
```

Con la función show() podremos mostrar el contenido de un dataframe

Aplicando transformaciones

Vamos a aplicar nuestra primera **transformación** a los datos. Es importante entender que aquí Spark **no hace nada**, solo toma nota de lo que hay que hacer.

```
df_mayores = df.filter("edad >= 18")
```

Usamos **filter()** para filtrar solo los registros cuya edad sea mayor de 18 años

```
df_mayores.explain()
```

Con **explain()** podemos saber cuál es el plan de trabajo

```
== Physical Plan ==
*(1) Filter (isnotnull(edad#1L) AND (edad#1L >= 18))
+- *(1) Scan ExistingRDD[nombre#0,edad#1L]
```

Internamente Spark trabaja con **RDDs**.

Aquí vemos los pasos que Spark realizará para realizar la operación solicitada. A medida que vayamos añadiendo transformaciones realizará **optimizaciones** sobre este plan

Acciones: ejecutando el plan de trabajo

Cuando realizamos una **acción** es cuando pone en marcha el plan de trabajo. En este caso vamos a usar la acción **show(),** que muestra los resultados

```
df_mayores.show()
```

```
+------+----+
|nombre|edad|
+------+----+
|Carlos|  22|
| Marta|  35|
+------+----+
```

2 DATAFRAMES: INGESTA Y TRANSFORMACIÓN

2.2 INGESTA DE DATOS

En el ejemplo anterior creamos los datos directamente en el código, pero lo habitual es obtenerlos de alguna fuente externa.

Spark puede recoger los datos desde diferentes fuentes como CSV, JSON o Parquet.

Esquema

Algo importante antes de leer los datos es el **esquema**, que básicamente se puede definir como **el contrato que define la estructura de los datos**.

El esquema le dice tres cosas al motor sobre cada columna:

- **Nombre** (cómo se llama)
- **Tipo de datos** (qué es: `DateType`, `IntegerType`, `ArrayType`, ...)
- **Nullability** (¿puede estar vacío?)

Si no indicamos el esquema, Spark lo inferirá a partir de los datos, pero indicarlo explícitamente tiene una serie de ventajas:

- **Rendimiento**: si no lo definimos, Spark tendrá que leer el archivo completo antes de cargar los datos para intentar averiguar qué tipo de datos son
- **Seguridad de tipos**: evitar que una columna se cargue con un tipo erróneo (p.e. si en una columna de precios hay un registro con valor 10.5€ en lugar de 10.5)
- **Optimización**: Spark reserva memoria de forma más eficiente si sabe, por ejemplo, que una columna es `Byte` (1 byte) y no un `Long` (8 bytes)

Para definir el esquema, usaremos los siguientes objetos del módulo `pyspark.sql.types`:

- **`StructType`**: representa la **fila** completa. Es una colección (lista) de campos
- **`StructField`**: representa una **columna** individual

Los tipos de datos que hay son:

Equivalente en Python	Tipo PySpark	Descripción
`str`	**`StringType()`**	Texto libre
`int`	**`IntegerType()`**	Entero estándar (32 bits)
`int (grande)`	**`LongType()`**	Entero grande (64 bits)
`float`	**`DoubleType()`**	Decimales (por defecto en Spark)
`decimal`	**`DecimalType()`**	Finanzas (precisión exacta)
`datetime`	**`TimestampType()`**	Fecha y hora
`list`	**`ArrayType(T)`**	Lista de elementos de tipo T
`dict`	**`MapType(K, V)`**	Clave-valor

Veamos cómo sería el código para definir el esquema:

```
from pyspark.sql.types import StructType, StructField
from pyspark.sql.types import StringType, IntegerType, BooleanType

schema = StructType([
    StructField("id", IntegerType(), False),
    StructField("nombre", StringType(), True),
    StructField("es_cliente", BooleanType(), True)
])
```

Tenemos que importar los tipos de datos que usaremos

Instanciamos el objeto `StructType`.

Para cada campo definimos el nombre, el tipo de datos que contendrá y si puede ser nulo (`True`) o no (`False`)

```
schema.fields
```

```
[StructField('id', IntegerType(), False),
 StructField('nombre', StringType(), True),
 StructField('es_cliente', BooleanType(), True)]
```

Podemos usar la propiedad `Fields` para ver la estructura del esquema

Aunque el ejemplo anterior era una tabla plana, el esquema puede ser **anidado** conteniendo datos complejos (tipo JSON o NoSQL como MongoDB)

```
direccion_schema = StructType([
    StructField("calle", StringType(), True),
    StructField("cp", StringType(), True)
])

main_schema = StructType([
    StructField("id", IntegerType(), True),
    StructField("nombre", StringType(), True),
    StructField("direccion", direccion_schema, True)
])
```

Creamos primero el esquema del tipo complejo

Y luego lo indicamos como tipo de datos del tipo principal

```
root
 |-- id: integer (nullable = true)
 |-- nombre: string (nullable = true)
 |-- direccion: struct (nullable = true)
 |    |-- calle: string (nullable = true)
 |    |-- cp: string (nullable = true)
```

Para cargar datos directamente en el *dataframe* debemos utilizar la función **createDataFrame()**, a la que opcionalmente se le puede pasar el esquema.

```
data = [
    (1, "Juan Perez", ("Calle Gran Vía 25", "28013")),
    (2, "Maria Lopez", ("Av. Diagonal 100", "08018")),
    (3, "Carlos Ruiz", None)
]

df = spark.createDataFrame(data, schema=main_schema)
```

Usamos el parámetro `schema` para indicar el esquema al crear el dataframe

```
df.printSchema()
```

Con la función `printSchema()` podremos visualizar el esquema del dataframe

```
root
 |-- id: integer (nullable = true)
 |-- nombre: string (nullable = true)
 |-- direccion: struct (nullable = true)
 |    |-- calle: string (nullable = true)
 |    |-- cp: string (nullable = true)
```

Si tengo campos anidados, podré acceder a ellos utilizando la notación del punto

La transformación `select()` filtra por columnas

```
df.select("nombre", "direccion.calle").show()
```

```
+----------+----------------+
|    nombre|          calle|
+----------+----------------+
| Juan Perez|Calle Gran Vía 25|
|Maria Lopez| Av. Diagonal 100|
|Carlos Ruiz|            NULL|
+----------+----------------+
```

La acción `show()` muestra el dataframe

INGESTA DE DATOS EN SPARK

Víctor J. González

Para cargar datos desde un fichero debemos utilizar el objeto **`spark.read`**, que nos da acceso a la API **`DataFrameReader`**.

La sintaxis general para leer un fichero es:

```
df = ( spark.read
          .format("tipo_archivo")
          .option("clave", "valor")
          .schema(mi_esquema)
          .load("ruta/al/archivo")
     )
```

Formato de la fuente de datos: csv, json, parquet, orc, text, jdbc

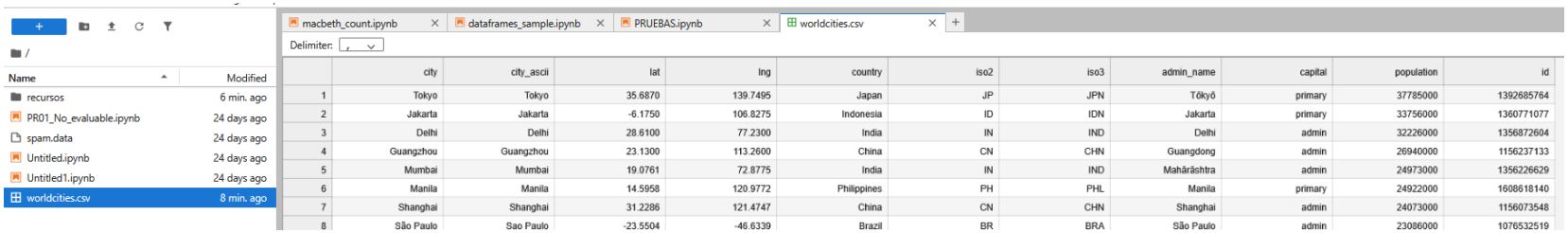
Configuraciones específicas del formato (p.e. separador para CSV o multiline para JSON)

Si hemos creado manualmente el esquema lo indicamos aquí

Ruta al fichero, puede ser local (`file://`) o distribuida (`hdfs://`, `s3a://`)

Carga de datos de CSV.

Vamos a leer el fichero worldcities.csv



	city	city_ascii	lat	lng	country	iso2	iso3	admin_name	capital	population	id
1	Tokyo	Tokyo	35.6870	139.7495	Japan	JP	JPN	Tōkyō	primary	37785000	1392685764
2	Jakarta	Jakarta	-6.1750	106.8275	Indonesia	ID	IDN	Jakarta	primary	33756000	1360771077
3	Delhi	Delhi	28.6100	77.2300	India	IN	IND	Delhi	admin	32226000	1356872604
4	Guangzhou	Guangzhou	23.1300	113.2600	China	CN	CHN	Guangdong	admin	26940000	1156237133
5	Mumbai	Mumbai	19.0761	72.8775	India	IN	IND	Mahārāshtra	admin	24973000	1356226629
6	Manila	Manila	14.5958	120.9772	Philippines	PH	PHL	Manila	primary	24922000	1608618140
7	Shanghai	Shanghai	31.2286	121.4747	China	CN	CHN	Shanghai	admin	24073000	1156073548
8	São Paulo	Sao Paulo	-23.5504	-46.6339	Brazil	BR	BRA	São Paulo	admin	23086000	1076532519

```
!head -n2 ./worldcities.csv
```

```
"city","city_ascii","lat","lng","country","iso2","iso3","admin_name","capital","population","id"
"Tokyo","Tokyo","35.6870","139.7495","Japan","JP","JPN","Tōkyō","primary","37785000","1392685764"
```

Comenzamos analizando la estructura de los datos para crear manualmente el esquema.

World cities database

Accurate and up-to-date database of the world's cities and towns

https://www.kaggle.com/datasets/juanmah/world-cities

Una vez comprendidos los datos, creamos el esquema:

```
from pyspark.sql.types import StructType, StructField, StringType, DoubleType, LongType

schema_worldcities = StructType([
    StructField("city", StringType(), True),
    StructField("city_ascii", StringType(), True),
    StructField("lat", DoubleType(), True),
    StructField("lng", DoubleType(), True),
    StructField("country", StringType(), True),
    StructField("iso2", StringType(), True),
    StructField("iso3", StringType(), True),
    StructField("admin_name", StringType(), True),
    StructField("capital", StringType(), True),
    StructField("population", LongType(), True),
    StructField("id", LongType(), True)
])
```

Y procedemos a cargar los datos:

```
df_cities = ( spark.read
                .format("csv")
                .schema(schema_worldcities)
                .option("header", "true")
                .option("quote", "\"")
                .load("./worldcities.csv")
            )

df_cities.printSchema()
df_cities.show(5)
```

Como el fichero tiene encabezado se lo indicamos

En el fichero todos los campos están rodeados de comillas dobles. Con `quote` se lo indicamos de forma que las omita y procese únicamente lo que hay entre comillas.

```
"city","city_ascii","lat",
"Tokyo","Tokyo","35.6870",
```

Con show(5) muestro las 5 primeras filas del *dataframe*

Comprobamos que el esquema del *dataframe* se corresponde con lo que le indicamos

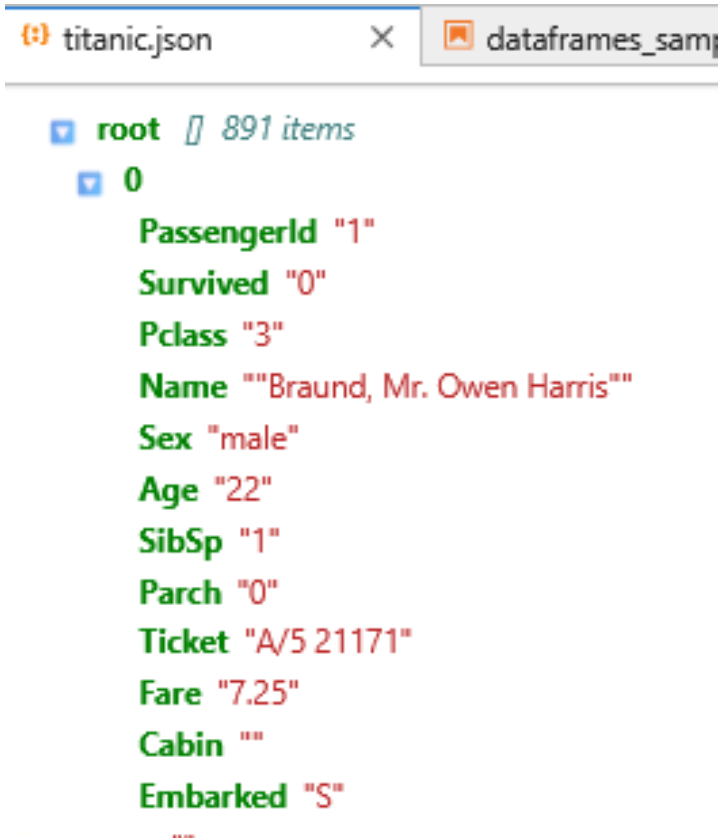
```
root
 |-- city: string (nullable = true)
 |-- city_ascii: string (nullable = true)
 |-- lat: double (nullable = true)
 |-- lng: double (nullable = true)
 |-- country: string (nullable = true)
 |-- iso2: string (nullable = true)
 |-- iso3: string (nullable = true)
 |-- admin_name: string (nullable = true)
 |-- capital: string (nullable = true)
 |-- population: long (nullable = true)
 |-- id: long (nullable = true)
```

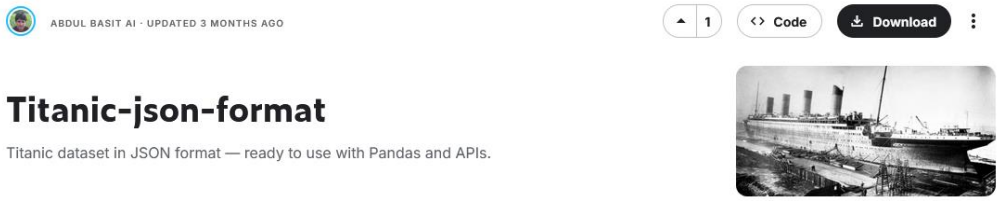
Y podemos ver los datos ya cargados

```
+---------+----------+-------+--------+---------+----+----+----------+-------+----------+----------+
|     city|city_ascii|    lat|     lng|  country|iso2|iso3|admin_name|capital|population|        id|
+---------+----------+-------+--------+---------+----+----+----------+-------+----------+----------+
|    Tokyo|     Tokyo| 35.687|139.7495|    Japan|  JP| JPN|      Tōkyō|primary|  37785000|1392685764|
|  Jakarta|   Jakarta| -6.175|106.8275|Indonesia|  ID| IDN|   Jakarta|primary|  33756000|1360771077|
|    Delhi|     Delhi|  28.61|   77.23|    India|  IN| IND|     Delhi|  admin|  32226000|1356872604|
|Guangzhou| Guangzhou|  23.13|  113.26|    China|  CN| CHN| Guangdong|  admin|  26940000|1156237133|
|   Mumbai|    Mumbai|19.0761| 72.8775|    India|  IN| IND|Mahārāshtra|  admin|  24973000|1356226629|
+---------+----------+-------+--------+---------+----+----+----------+-------+----------+----------+
only showing top 5 rows
```

Carga de datos JSON

Vamos a ver cómo cargar datos de un JSON, en esta ocasión del *dataset* del Titanic





https://www.kaggle.com/datasets/engrbasit62/titanic-json-format

Comenzamos creando el esquema

```python
# Definición del esquema para el dataset Titanic
schema_titanic = StructType([
    StructField("PassengerId", IntegerType(), True),
    StructField("Survived", IntegerType(), True),    # 0 o 1
    StructField("Pclass", IntegerType(), True),      # 1, 2 o 3
    StructField("Name", StringType(), True),
    StructField("Sex", StringType(), True),
    StructField("Age", DoubleType(), True),          # Double porque hay edades fraccionarias (ej: 0.42)
    StructField("SibSp", IntegerType(), True),       # Esposos/Hermanos a bordo
    StructField("Parch", IntegerType(), True),       # Padres/Hijos a bordo
    StructField("Ticket", StringType(), True),
    StructField("Fare", DoubleType(), True),         # Tarifa (dinero)
    StructField("Cabin", StringType(), True),
    StructField("Embarked", StringType(), True)      # Puerto de embarque (C, Q, S)
])
```

Y cargamos los datos

```
df_titanic = ( spark.read
                    .format("json")
                    .schema(schema_titanic)
                    .option("multiline", "true")
                    .load("titanic.json")
             )

df_titanic.printSchema()
df_titanic.show(5)
```

Le tenemos que avisar a Spark de que el fichero no es JSONL y tiene varias líneas por registro

Por defecto, Spark espera el formato **JSONL** (JSON Lines) donde:

- Cada línea es un objeto JSON completo y válido
- No hay corchetes envolviendo todo el archivo
- No hay comas al final separando los objetos

```
+-----------+--------+------+--------------------+------+----+-----+-----+----------------+----+-----+--------+
|PassengerId|Survived|Pclass|                Name|   Sex| Age|SibSp|Parch|          Ticket|Fare|Cabin|Embarked|
+-----------+--------+------+--------------------+------+----+-----+-----+----------------+----+-----+--------+
|       NULL|    NULL|  NULL|"Braund, Mr. Owen...|  male|NULL| NULL| NULL|       A/5 21171|NULL|     |       S|
|       NULL|    NULL|  NULL|"Cumings, Mrs. Jo...|female|NULL| NULL| NULL|        PC 17599|NULL|  C85|       C|
|       NULL|    NULL|  NULL|"Heikkinen, Miss....|female|NULL| NULL| NULL|STON/O2. 3101282|NULL|     |       S|
|       NULL|    NULL|  NULL|"Futrelle, Mrs. J...|female|NULL| NULL| NULL|          113803|NULL| C123|       S|
|       NULL|    NULL|  NULL|"Allen, Mr. Willi...|  male|NULL| NULL| NULL|          373450|NULL|     |       S|
+-----------+--------+------+--------------------+------+----+-----+-----+----------------+----+-----+--------+
```

INGESTA DE DATOS EN SPARK

Víctor J. González

Cuando leemos ficheros puede que encontremos **datos corruptos o malformados**.

Debemos indicarle a Spark como actuar ante estos datos mediante `.option("mode", "mode_name")`

Los modos que hay son:

- **PERMISSIVE**:
  - Es el modo por defecto
  - Si hay un dato que no encaja con el tipo esperado **no falla ni elimina la fila**
  - Convierte el valor corrupto a `null` y carga el resto de la fila
  - El problema es que se puede llenar el DataFrame de nulos silenciosamente y estropear cálculos sin darnos cuenta.
  - En este modo podemos configurar una columna especial llamada `_corrupt_record` para que Spark guarde ahí el dato original sucio y se pueda revisar luego.

Ejemplo de cómo guardar los datos corruptos en una columna aparte

```
schema = StructType([
    StructField("id", IntegerType(), True),
    StructField("nombre", StringType(), True),
    StructField("edad", IntegerType(), True),
    StructField("_corrupt_record", StringType(), True)
])

df = ( spark.read
                .format("csv")
                .option("header", "true")
                .schema(schema)
                .option("mode", "PERMISSIVE")
                .option("columnNameOfCorruptRecord", "_corrupt_record")
                .load("datos_sucios.csv")
     )
```

En esta columna guardaremos los datos corruptos

Debemos usar la opción `columnNameOfCorrruptRecord`

- **DROPMALFORMED**:
  - Si una fila tiene datos corruptos **la ignora por completo**
  - Con esto aseguramos que los datos estén limpios, pero en un *dataset* con muchos datos mal formados perderemos muchas líneas y Spark no nos avisará explícitamente.
- **FAILFAST**:
  - Al primer error que encuentre, Spark **detiene todo el proceso** inmediatamente y lanza una excepción
  - Garantiza al 100% la calidad de los datos
  - Se usa en sistemas críticos (p.e. transacciones bancarias) donde no se puede permitir ni un error o dato sucio.

2 DATAFRAMES: INGESTA Y TRANSFORMACIÓN

2.3 OPERACIONES BÁSICAS

OPERACIONES BÁSICAS

Víctor J. González

Una vez cargados los datos en el DataFrame, lo habitual es prepararlos antes del análisis seleccionando columnas, filtrando filas, corrigiendo valores y creando nuevas métricas.

Es importante recordar que los DataFrames son **inmutables**, es decir, las operaciones no modifican el original, sino que de devuelven un nuevo DataFrame con los cambios aplicados.

SELECCIÓN: SELECT

Equivale al SELECT de SQL y sirve para **seleccionar una serie de columnas** y descartar el resto.

Por convención, las funciones de PySpark se importan como **F**

```
from pyspark.sql import functions as F
from pyspark.sql.functions import col, lit

df_cities.select("city", "population").show(3)
```

```
+-------+----------+
|   city|population|
+-------+----------+
|  Tokyo|  37785000|
|Jakarta|  33756000|
|  Delhi|  32226000|
+-------+----------+
only showing top 3 rows
```

Indico las columnas con las que me quiero separar separadas por comas

En lugar de indicar el nombre de la columna, puedo usar la función `col()` que añade más funcionalidades

```
df_cities.select(col("city"), col("population")/1000000).show(3)
```

```
+-------+----------------------+
|   city|(population / 1000000)|
+-------+----------------------+
|  Tokyo|                37.785|
|Jakarta|                33.756|
|  Delhi|                32.226|
+-------+----------------------+
only showing top 3 rows
```

Puedo realizar **operaciones** con todos los elementos de la columna. En este ejemplo, mostrar los habitantes por millones

```
df_cities.select(col("city"), col("population").alias("población")).show(3)
```

```
+-------+---------+
|   city|población|
+-------+---------+
|  Tokyo| 37785000|
|Jakarta| 33756000|
|  Delhi| 32226000|
+-------+---------+
only showing top 3 rows
```

Puedo aplicar la función **`alias()`** para **cambiar el nombre** de la columna

OPERACIONES BÁSICAS

Víctor J. González

```
df_cities.select(col("city"), col("population").cast("float")).show(3)
```

```
+-------+----------+
|   city|population|
+-------+----------+
|  Tokyo|  3.7785E7|
|Jakarta|  3.3756E7|
|  Delhi|  3.2226E7|
+-------+----------+
only showing top 3 rows
```

Otra opción es usar la función `cast()` para **cambiar el tipo de datos** de la columna

```
df_cities.select(col("city"), col("population")>10000000).show(3)
```

```
+-------+-----------------------+
|   city|(population > 10000000)|
+-------+-----------------------+
|  Tokyo|                   true|
|Jakarta|                   true|
|  Delhi|                   true|
+-------+-----------------------+
only showing top 3 rows
```

También podemos **generar una columna True/False** según el valor correspondiente cumpla una condición

RENOMBRADO: `withColumnRenamed()`

Renombra una columna del DataFrame

```
df_cities.withColumnRenamed("Population", "Población").show(3)
```

```
+-------+---------+------+--------+---------+----+----+----------+-------+---------+-----
|   city|city_ascii|   lat|     lng|  country|iso2|iso3|admin_name|capital|Población|
+-------+---------+------+--------+---------+----+----+----------+-------+---------+-----
|  Tokyo|     Tokyo|35.687|139.7495|    Japan|  JP| JPN|     Tōkyō|primary| 37785000|13926
|Jakarta|   Jakarta|-6.175|106.8275|Indonesia|  ID| IDN|   Jakarta|primary| 33756000|13607
|  Delhi|     Delhi| 28.61|   77.23|    India|  IN| IND|     Delhi|  admin| 32226000|13568
+-------+---------+------+--------+---------+----+----+----------+-------+---------+-----
only showing top 3 rows
```

FILTRADO DE FILAS: `where()` y `filter()`

Ambas funciones son sinónimos exactos y se pueden usar indistintamente.

Filtran los registros del DataFrame según un criterio.

Los operadores Python no funcionan, sino que se deben usar los operadores **&** (AND), **|** (OR) y **~** (NOT)

Cuando hay condiciones compuestas es obligatorio el uso de paréntesis

```
df_cities.filter( col("country") == "Spain" ).show(3)
```

```
+---------+----------+------+------+-------+----+----+----------+-------+----------+-----
|     city|city_ascii|   lat|   lng|country|iso2|iso3|admin_name|capital|population|
+---------+----------+------+------+-------+----+----+----------+-------+----------+-----
|   Madrid|    Madrid|40.4169|-3.7033|  Spain|  ES| ESP|    Madrid|primary|   6211000|17246
|Barcelona| Barcelona|41.3833| 2.1833|  Spain|  ES| ESP| Catalonia|  admin|   4800000|17245
| Valencia|  Valencia|  39.47|-0.3764|  Spain|  ES| ESP|  Valencia|  admin|   1595000|17249
+---------+----------+------+------+-------+----+----+----------+-------+----------+-----
only showing top 3 rows
```

Ejemplo de condición compuesta

```
(df_cities
    .filter( (col("country") == "Spain") & ( col("population") > 5000000 ) )
    .select( "city", "country", "population" )
    .show(3)
)
```

```
+------+-------+----------+
|  city|country|population|
+------+-------+----------+
|Madrid|  Spain|   6211000|
+------+-------+----------+
```

CREACIÓN DE COLUMNAS: `withColumn()`

Este método sirve para dos cosas:

- Crear una columna nueva (si el nombre no existe)
- Sobrescribir una columna existente (si el nombre existe)

```
(df_cities
     .withColumn("million_pop", col("population")/1000000)
     .select("city", "population", "million_pop")
     .show(3)
)
```

```
+-------+----------+-----------+
|   city|population|million_pop|
+-------+----------+-----------+
|  Tokyo|  37785000|     37.785|
|Jakarta|  33756000|     33.756|
|  Delhi|  32226000|     32.226|
+-------+----------+-----------+
only showing top 3 rows
```

Si queremos que todas las columnas tengan un valor constante tenemos que utilizar la función `lit()`

```
(df_cities
     .withColumn("year", lit(2025))
     .select("city", "population", "year")
     .show(3)
)
```

La función `lit` la importamos de `pyspark.sql.functions`

```
+-------+----------+----+
|   city|population|year|
+-------+----------+----+
|  Tokyo|  37785000|2025|
|Jakarta|  33756000|2025|
|  Delhi|  32226000|2025|
+-------+----------+----+
only showing top 3 rows
```

OPERACIONES BÁSICAS

Víctor J. González

Podemos realizar diferentes tipos de operaciones en `withColumn()`

1. **Matemáticas y redondeo**

Para columnas numéricas la clase F dispone de funciones para:

- **Redondeo**: `ceil` (techo), `floor` (suelo), `round` (redondeo), `bround` (redondeo bancario al par más cercano para reducir el sesgo en sumas grandes.
- **Logaritmos y exponenciales**: `log` (logaritmo natural), `log10` (logaritmo base 10), `exp` (exponencial e^x), `factorial` (factorial de un entero).
- **Comparación entre columnas**: `max` y `min` (máximo / mínimo de una columna verticalmente), `greatest(col1, col2, …)` y `least (col1, col2, …)` (máximo / mínimo entre varias columnas de una misma fila)
- **Trigonometría**: `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `degrees` (radianes a grados), `radians` (grados a radianes)
- **Aleatoriedad**: `rand` (número entre 0 y 1), `randn` (número aleatorio de distribución normal o gausiana)

2. **Manipulación de fechas y tiempo**

Tenemos las siguientes funciones para fechas:

- `to_date` / `to_timestamp`: convertir strings a fechas
- `datediff`: restar fechas (devuelve número de días)
- `date_add` / `date_sub`: sumar/restar días a una fecha
- `year`, `mont`, `dayofmonth`: extraer partes de una fecha

3. **Manipulación de cadenas**

Para el trabajo con cadenas hay un montón de funciones:

- **Limpieza y formato**:
  - **`trim`**`():` elimina espacios blancos a ambos lados
  - **`ltrim`**`() /` **`rtrim`**`():` elimina espacios solo a izquierda o derecha
  - **`upper`**`() /` **`lower`**`():` convierte a mayúsculas/minúsculas
  - **`initcap`**`():` `tipo` título
- **Unión y división**:
  - **`concat`**`(col1, col2, …):` une columnas sin separador. Si alguna columna es null, el resultado será null.
  - **`concat_ws`**`(sep, col1, col2, …):` concatena columnas usando un separador. Si hay nulos se los salta.
  - **`split`**`(col, sep):` divide un string en una lista

OPERACIONES BÁSICAS

Víctor J. González

- **Extracción y longitud:**
  - **`length`**`():` devuelve el número de caracteres
  - **`substring`**`(col, pos, len):` extrae una parte del texto. ***Ojo: los índices comienzan por 1***.
  - **`instr`**`(col, sub):` busca la posición de una subcadena (0 si no la encuentra.
- **Reemplazo y expresiones regulares**:
  - **`translate`**`(col, “ABC”, “123”):` reemplazo carácter a carácter (A->1, B->2, ..) Muy rápido para limpiezas simples
  - **`regexp_replace`**`(col, pattern, replacement):` busca un patrón regexp y lo reemplaza
  - **`regexp_extract`**`(col, pattern, goup_idx):` extrae algo específico que cumpla un patrón (p.e. el dominio de un email)
- **Relleno**:
  - **`lpad`** / **`rpad`**`(col, len, pad):` rellena por la izquierda o la derecha

4. **Manejo de nulos**

Para el manejo de nulos Spark dispone de la función **`coalesce()`**.

Esta función se puede ver como una cascada de opciones que recibe varias columnas o valores y **devuelve el primero que no sea nulo**.

Permite fusionar información dispersa en varias columnas para crear una columna maestra

```
df = df.withColumn("email_final",
    coalesce(
        col("email_personal"),
        col("email_trabajo"),
        lit("No disponible")
    )
)
```

Ejemplo: quiero hacer una campaña de envío de correos, pero de algunos usuarios tengo solo el personal, de otros solo el del trabajo y de otros ninguno.

3. **Tipos complejos**

PySpark puede manejar **estructuras de datos** dentro de una sola celda.

Para convertir datos planos en lisas tenemos las siguientes funciones:

- **`split`**`(col, sep)`: convierte una cadena de texto en un array cortando por el delimitador.
- **`array`**`(col1, col2, …)`: junta los valores de varias columnas del mismo registro en una sola lista.

```
( df_titanic
      .withColumn('sep_name', F.split(col("name"), " "))
      .select("sep_name", "Name")
      .show(3, truncate=False)
)
```

```
+------------------------------------------------------------+--------------------------
|sep_name                                                    |Name
+------------------------------------------------------------+--------------------------
|["Braund,, Mr., Owen, Harris"]                              |"Braund, Mr. Owen Harris"
|["Cumings,, Mrs., John, Bradley, (Florence, Briggs, Thayer)"]|"Cumings, Mrs. John Bradley
|["Heikkinen,, Miss., Laina"]                                |"Heikkinen, Miss. Laina"
+------------------------------------------------------------+--------------------------
```

OPERACIONES BÁSICAS

Víctor J. González

Una vez que tenemos una lista o array, la pregunta es ¿qué podemos hacer con él?

- **`size`**`(col)`: devuelve el número de elementos (longitud)
- **`array_contains`**`(col, value)`: devuelve True si el valor está en la lista
- **`col[i]`:** permite acceder a un elemento específico (ojo, aquí el primer elemento es el índice 0)

```python
# 1. Filtrar productos que tengan la etiqueta "urgente"
df.filter(F.array_contains(col("array_etiquetas"), "urgente"))

# 2. Crear columna con el primer elemento de la lista
df = df.withColumn("etiqueta_principal", col("array_etiquetas")[0])

# 3. Filtrar listas vacías
df.filter(F.size(col("array_etiquetas")) > 0)
```

La otra opción es **descomponer el array** con la función **`explode()`**.

Esta función toma un array en una fila y **genera una nueva fila por cada elemento** del array.

```python
from pyspark.sql.functions import explode

# Supongamos df:
# +---+------------------+
# | id|   array_etiquetas|
# +---+------------------+
# |  1| ["cine", "cena"] |
# +---+------------------+

df_explotado = df.withColumn("etiqueta_individual", explode(col("array_etiquetas")))

# Resultado df_explotado:
# +---+------------------+------------------+
# | id|   array_etiquetas|etiqueta_individual|
# +---+------------------+------------------+
# |  1| ["cine", "cena"] |              cine|
# |  1| ["cine", "cena"] |              cena|
# +---+------------------+------------------+
```

Ojo, si tenemos listas con muchos elementos el DataFrame crecerá exponencialmente

OPERACIONES BÁSICAS

Víctor J. González

Si tenemos varias columnas con arrays, podremos realizar operaciones entre ellos tratándolos como conjuntos:

- **`array_intersect`**`(col1, col2)`: elementos comunes
- **`array_union`**`(col1, col2)`: unión sin duplicados
- **`array_except`**`(col1, col2)`: resta de conjuntos (elementos que están en col1 y no en col2)
- **`array_distinct`**`(col)`: elimina duplicados dentro de un mismo array

```
# Escenario: Comparar respuestas correctas vs respuestas del alumno
# col("clave"): ["A", "B", "C"]
# col("alumno"): ["A", "C", "D"]

# ¿Cuáles acertó?
df = df.withColumn("aciertos", F.array_intersect(col("clave"), col("alumno")))
# Resultado: ["A", "C"]
```

OPERACIONES BÁSICAS

Víctor J. González

Por último, también podemos convertir el array en una cadena simple, por ejemplo, si queremos exportar a CSV el DataFrame.

- **`array_join`**`(col, delimiter)`: convierte lista a texto

```
df = df.withColumn("texto_final", F.array_join(col("array_etiquetas"), " - "))
```