



Módulo Profesional: Big Data Aplicado

SPARK

¿QUÉ ES APACHE SPARK?

Es un motor de procesamiento de datos en gran escala, diseñado para realizar operaciones distribuidas sobre grandes volúmenes de datos.

Necesita trabajar en un sistema donde pueda acceder a los datos de forma concurrente y distribuida, lo que significa que los datos deben estar distribuidos a través de múltiples nodos o máquinas dentro de un clúster.

¿QUÉ ES APACHE SPARK?

- Apache Spark es open-source y es un framework de computación. Requiere:
 - **Cluster manager**
 - Standalone – un gestor de clúster sencillo incluido en Spark
 - Apache Mesos – un gestor de clúster general
 - Hadoop YARN – gestor de recursos incluido Hadoop 2.
 - **Sistema de archivos distribuido**
 - Hadoop Distributed File System (HDFS) – **normalmente**
 - Cassandra
 - Amazon S3, Google GCS etc...

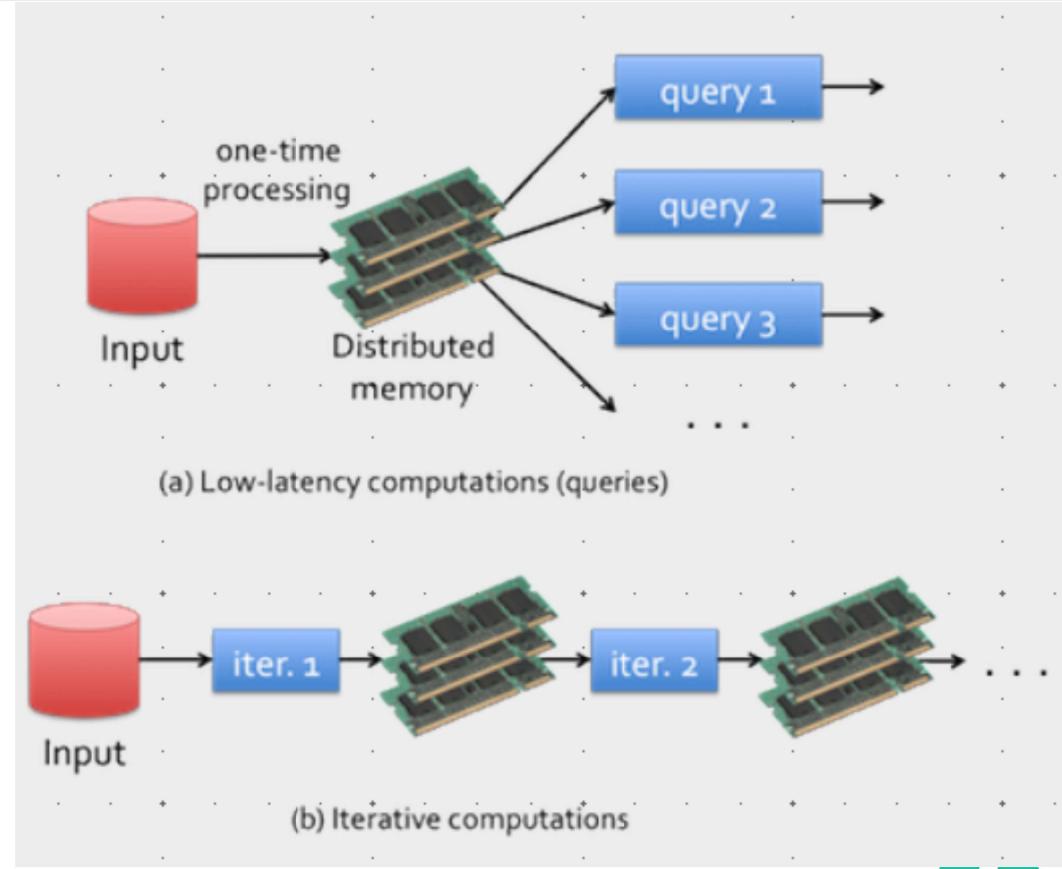
Soporta un modo pseudo-distribuido para desarrollo y test.
Sistema local de ficheros y un worker por cada CPU core.

¿QUÉ ES APACHE SPARK?

- Spark proporciona los siguientes beneficios principales:
 - **Computacion muy rápida**
 - Los datos se cargan en memoria distribuida (RAM) sobre servidores en clúster.
 - No necesita persistir pasos intermedios a disco.
 - **Muy accesible**
 - APIs de **Java**, **Scala**, **Python**, R o SQL
 - **Compatibilidad**
 - Con todo los sistemas de Hadoop existentes
 - **Práctico**
 - Shells Interactivas en Scala y Python (REPL)
 - **Mayor productividad**
 - Debido a estructuras de alto nivel que facilitan centrarse en los cálculos.

¿QUÉ ES APACHE SPARK?

- Apache Spark es una plataforma de computación cluster diseñada para **ser rápida, altamente accesible y de uso general**.
- Muy rápido



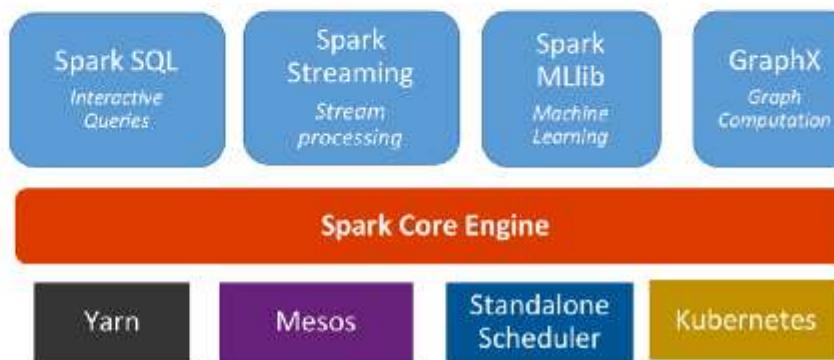
ARQUITECTURA

Spark

Unified, open source, parallel, data processing framework for Big Data Analytics

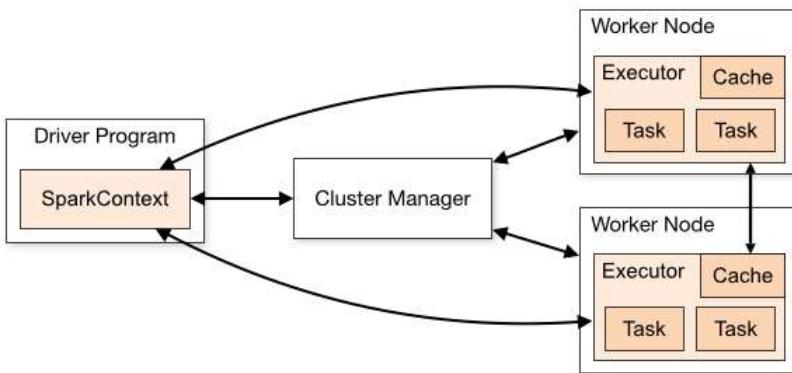
Spark Unifies:

- Batch Processing
- Real-time processing
- Stream Analytics
- Machine Learning
- Interactive SQL

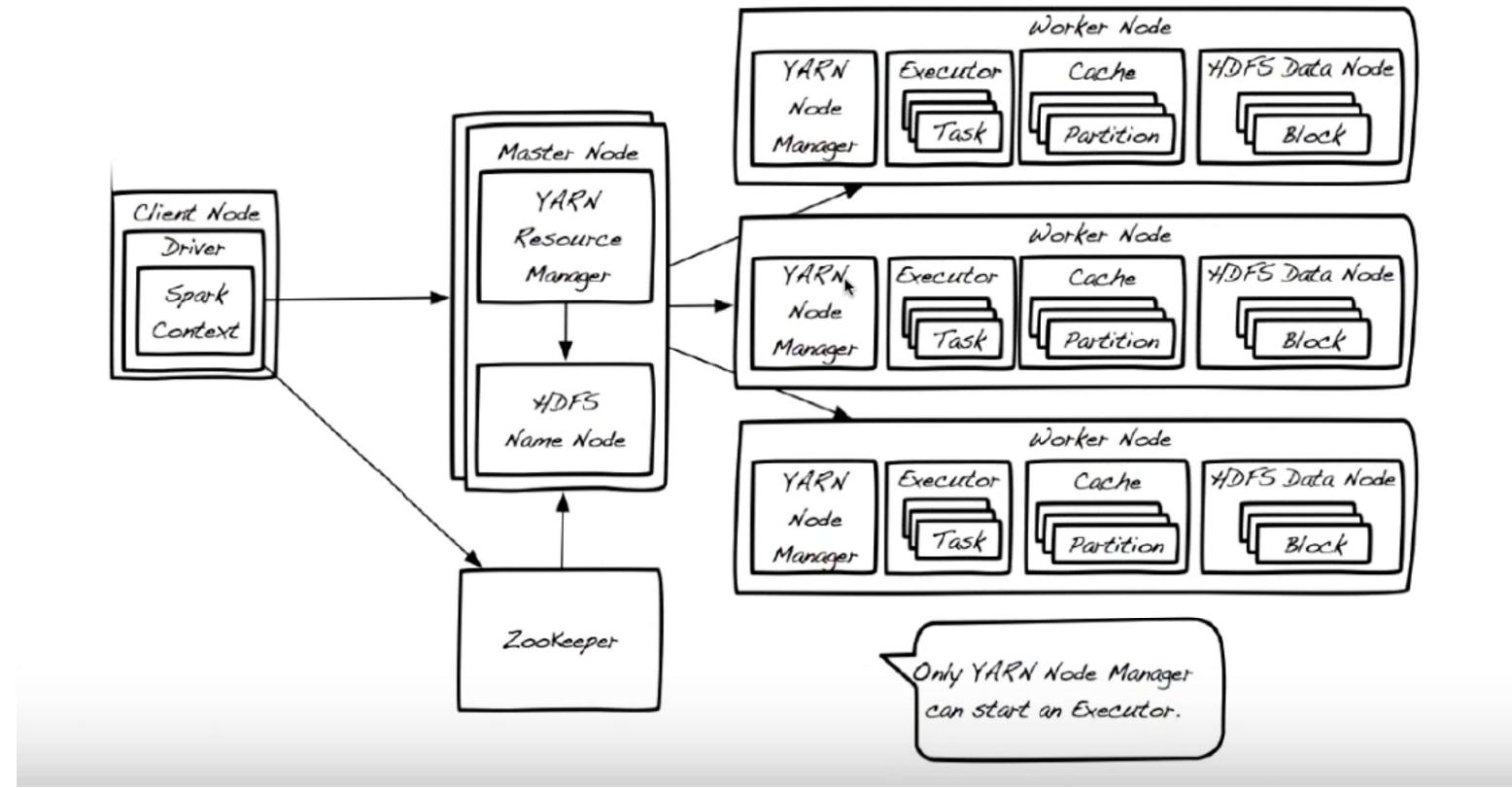


ARQUITECTURA: SPARK RUNTIME

- Arquitectura Maestro/Eslavo
- Coordinador central **driver** (own java process)
- Demonios en workers called **executor** (own java process)
- **Driver + executors = Spark application**
- La aplicación se lanza utilizando un gestor de cluster



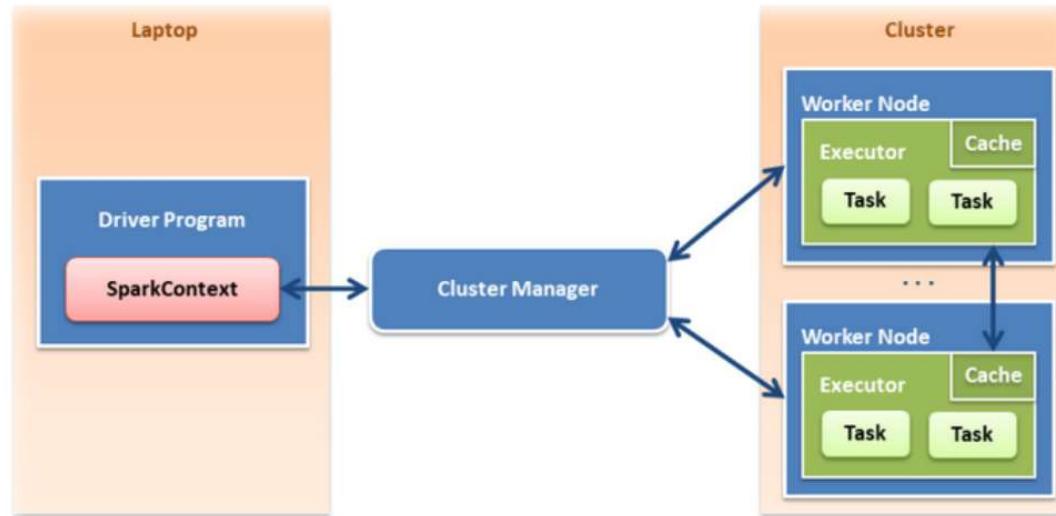
Spark YARN-based architecture



Conceptos Core Spark

- Toda aplicación Spark application debe tener un ***driver program*** que define los datos distribuidos datasets en un clúster y luego lanza las operaciones en paralelo.
- Driver puede ser tu propio programa o **Spark shell** con el tipo de operaciones que tú quieras ejecutar.
- Driver accede a Spark a través de **SparkSession object** que representa la conexión al clúster.

Apache Spark - DataFlow in client mode



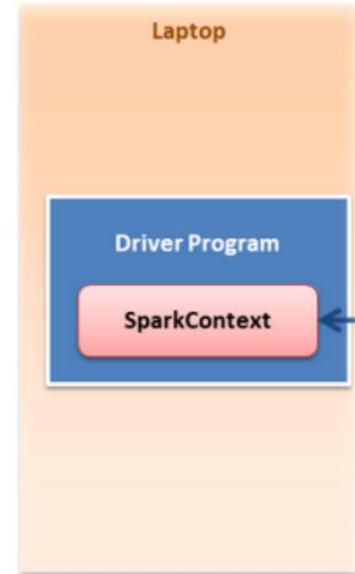
Apache Spark - DataFlow

- El código de Sparks se ejecuta en el Driver program, conectándose a través de *SparkSession*
- Nosotros definiremos todas fuentes y configuración con *SparkSession*, donde RDDs y otras estructuras que se crearán también.

```
import findspark
findspark.init('/opt/mapr/spark/spark-2.0.1')
import pyspark
from pyspark.sql import SparkSession

spark=SparkSession.builder.appName("variable_selection")\
    .config("spark.master", "yarn")\
    .config("spark.eventLog.enabled", "true")\
    .config("spark.executor.instances", "5")\
    .config("spark.executor.cores", "3")\
    .config("spark.executor.memory", "5g")\
    .getOrCreate()

df=spark.read.csv("hdfs:/data/databases/cars.csv")
df.show()
```



RDD – Resilient Distributed Dataset

- Es una **colección de datos inmutable y distribuida**, el cual está **particionado** en cluster.
- Facilita dos tipos de operaciones:
 - **Transformación**
 - Operaciones como filter(), map(), o union() en RDD que dan como resultado otro RDD.
 - **Evaluación “Lazy”**. Quiere decir que la evaluación no se ejecutan hasta que se realizan las acciones.
 - **Acciones**
 - Una acción es una operación como count(), first(), take(n), o collect() que lanzan los **cálculos**, devuelve un **valor** al Master o **escribe** en sistema de almacenamiento.
- Driver recuerda las transformaciones aplicadas al RDD (porque has construido un plan para ello), además si una partición se ha perdido, se puede volver a reconstruir en otra máquina del clúster otra vez.
 - Esto por lo que se llama “**Resilient**”

- RDDs se pueden crear de tres formas:
 1. Cargando de un dataset externo en HDFS, S3, GCS, fichero local, etc...

```
scala> val lines = sc.textFile("README.md")
```
 2. **Paralelizar** una colección de objetos existente en memoria del driver/worker

```
scala> val input = sc.parallelize(List(1, 2, 3, 4))
```
 3. Como resultado de una transformación a otro RDD

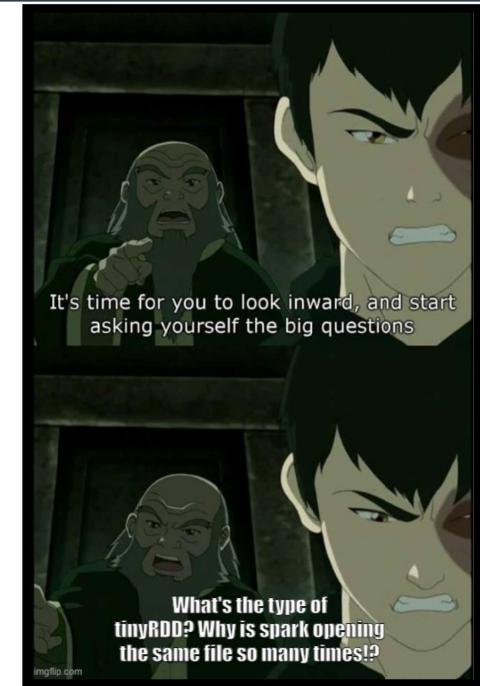
```
scala> val rdd2 = rdd1.filter(line => line.contains("error"))
```

Databricks

Se basa en apache Spark que nos permite realizar un procesamiento distribuido de datos, siendo una soluciones que permiten trabajar buscando la optimización del procesamiento y análisis de manera escalable y eficiente.

RDD – Introducción Apache Spark

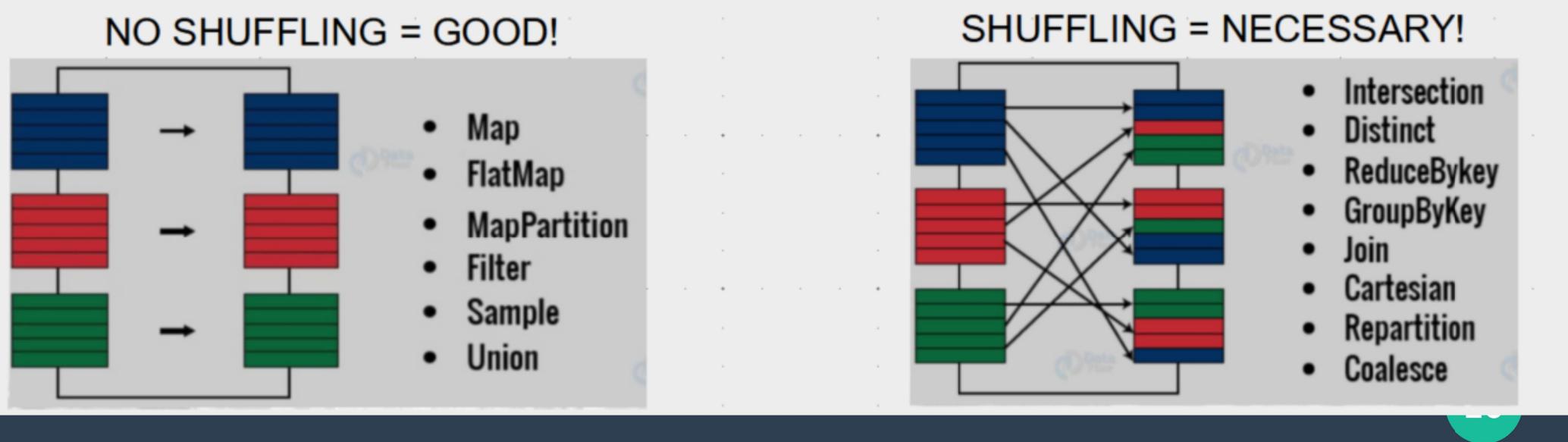
- Carga el notebook “0 – Introduction Apache Spark” en Databricks
 - *Ejecuta y analiza las siguientes secciones*
 - **1 - Basic RDD reads**
 - **2 - Parallelize RDDs & basic actions**
 - **Abrir <https://www.databricks.com/try-databricks#account>**
 - **Cuenta de correo**
<https://www.databricks.com/>



https://login.databricks.com/?dbx_source=www&itm=main-cta-login&tuuid=6aa42787-859a-4b6d-ba23-f62309435aa3

RDD – Resilient Distributed Dataset

- Operaciones sobre RDDs devuelven un nuevo RDD
- Transformaciones son **lazy**(*)
 - Sólo se calcula cuando requiere que devuelva un resultado a *Driver Program*
 - (*) Algunas transformaciones como *sortByKey* no son completamente lazy porque calcula inmediatamente los límites de las particiones (la ordenación real sí que es lazy)



RDD – Resilient Distributed Dataset: EXECUTION PLAN

Paso 1. Lee las líneas de un fichero en memoria

Paso 2. Separa las líneas en palabras → RDD de listas de strings (Transformación)

Paso 3. Map cada palabra tupla (word, 1) → RDD de tuplas de palabras, 1 (Transformación)

Paso 4. Reduce por agregación contando por palabras (word, count) → (Transformación)

Paso 5. Map (word, count) por (count, word) → (Transformación)

Paso 6. Sort by contador descendente → (Transformación)

Paso 7. Imprime las 5 palabras más frecuentes con sus respectivos contadores → Action

EJERCICIO

Open the notebook “0 - Introduction Apache Spark” in Databricks

- *Execute and analyze the following sections:*
 - **3 - Basic transformations**
 - **4 - Filter**
- *Let the fun begin!*
- *Try to complete the Hands on #1 - Wordcounting Frankie exercise*

Pares Key/Value RDD

- Spark permite operaciones con RDDs de pares (**key, value**)
- Operaciones que permiten actuar sobre cada clave en paralelo, reagrupar o agregar datos.
- Muchos de los formatos devuelven pares RDDs por key/value
- Paralelizar

```
val pairs = sc.parallelize (List((1,"a"), (2,"b"), (3,"c")))
```

Explicación:

sc.parallelize(...) crea un RDD a partir de una lista de tuplas.

List((1, "a"), (2, "b"), (3, "c")) es una lista de pares clave-valor:

(1, "a") (2, "b") (3, "c")

Pares Key/Value RDD

- Ejecutar transformaciones sobre RDDs

```
//pair RDD using the first word as the key  
val pairs = lines.map(x => (x.split(" "))(0), x))
```

Explicación:

- lines.map(...) → RDD de líneas de texto en un Pair RDD.
- x.split(" ")(0): x.split(" ") divide la línea en palabras separadas por espacios.
 (0) toma la primera palabra como clave.

Par (clave, valor):

- Clave: primera palabra de la línea.
- Valor: la línea completa.

Pares Key/Value RDD: JOIN

Combinar dos RDDs de pares clave-valor con claves en común, similar a una operación en SQL, siendo el valor final una tupla de valores:

(clave, (valor1, valor2))

```
> visits = sc.parallelize([ ("index.html", "1.2.3.4"),
   >                         ("about.html", "3.4.5.6"),
   >                         ("index.html", "1.3.3.1") ])

> pageNames = sc.parallelize([ ("index.html", "Home"),
   >                            ("about.html", "About") ])

> visits.join(pageNames)
# ("index.html", ("1.2.3.4", "Home"))
# ("index.html", ("1.3.3.1", "Home"))
# ("about.html", ("3.4.5.6", "About"))
```

Pares Key/Value RDD: COGROUPS

Permite agrupar los valores de múltiples RDDs de pares (clave, valor) que comparten la misma clave. Es como hacer un GROUP BY en SQL pero con múltiples datasets.

```
> visits = sc.parallelize([ ("index.html", "1.2.3.4"),
   ("about.html", "3.4.5.6"),
   ("index.html", "1.3.3.1") ])

> pageNames = sc.parallelize([ ("index.html", "Home"),
   ("about.html", "About") ])

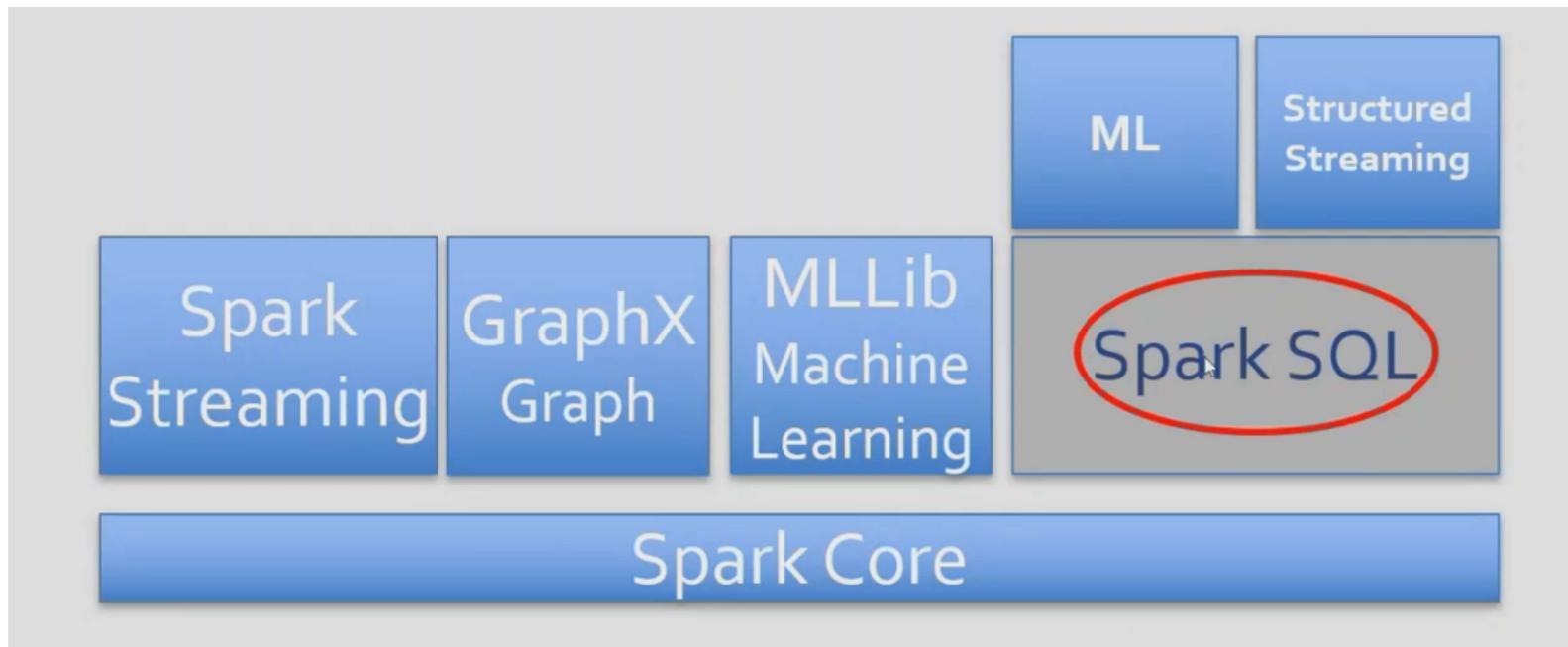
> visits.cogroup(pageNames)
# ("index.html", ([ "1.2.3.4", "1.3.3.1" ], [ "Home" ]))
# ("about.html", ([ "3.4.5.6" ], [ "About" ]))
```

Ejercicio

- Open the notebook “**0 – Introduction Apache Spark**” in Databricks
 - *Execute and analyze the following sections:*
 - **5 - CSV files with RDDs**
 - *Try the following challenge:*
 - **Hands on #2 - Joining the datasets**

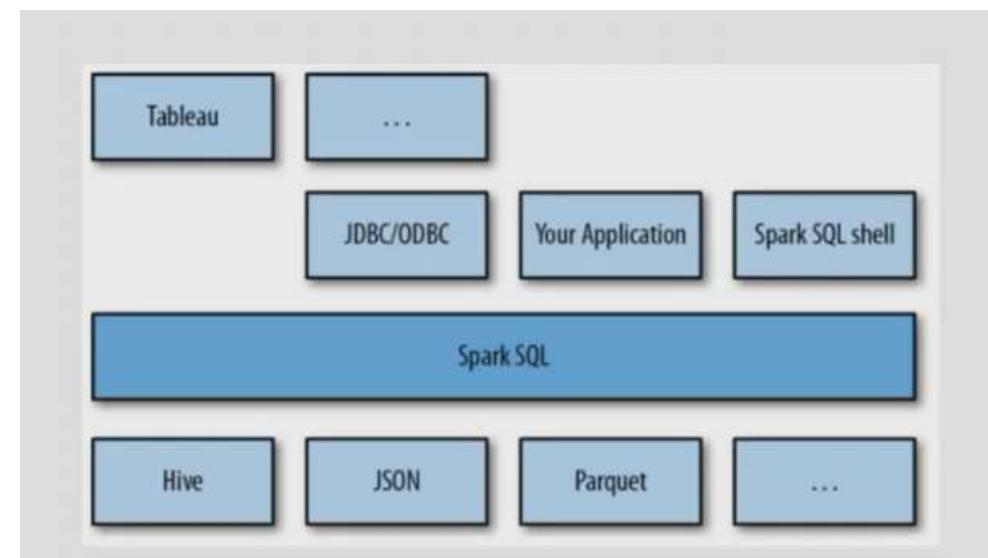
Introducción Spark SQL

- **Spark SQL** se lanzó por primera vez en Spark 1.0 (mayo, 2014)



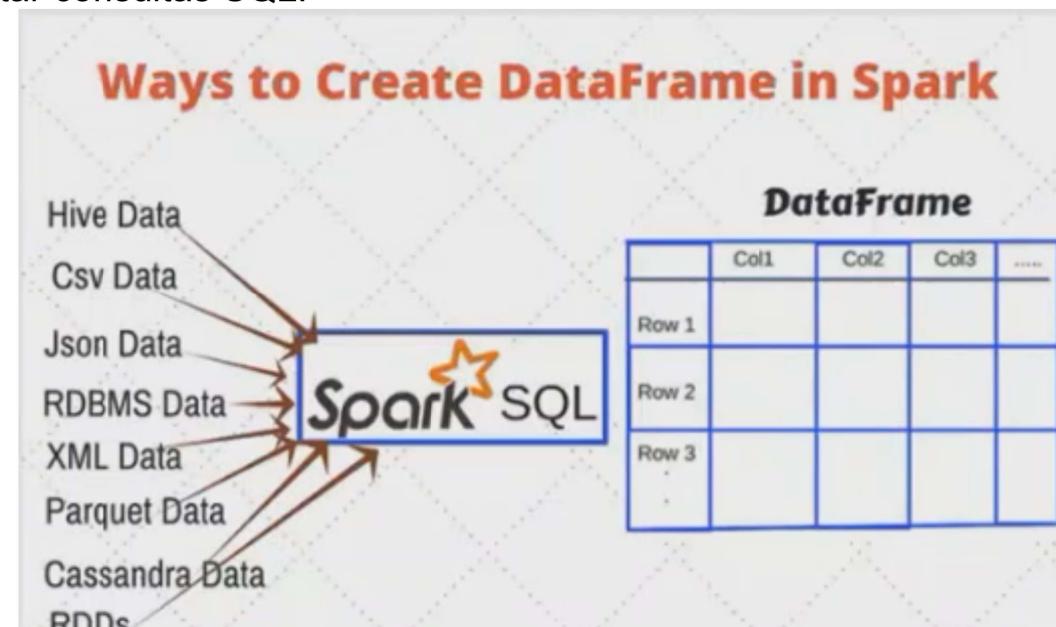
Introducción Spark SQL

- Es una interfaz para trabajar con datos estructurados (schema) y semi-estructurados.
- Spark SQL aplica vistas estructuradas a datos almacenados en distintos formatos.
- Tres capacidades principales:
 - Abstracción de DataFrame para datos estructurados.
 - Similar a las tablas relacionales de base de datos.
 - Lectura & Escritura en formatos estructurados (JSON,...)
 - Consulta de datos usando SQL dentro del programa Spark y desde herramientas externas usando JDBC/ODBC



DataFrames & Datasets

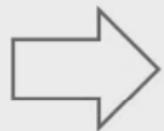
- Representa una colección distribuida (como RDDs)
- Añade un esquema de información no se encuentra en los RDDs
- Capacidad de almacenamiento más eficiente (Tungsten)
- Proporciona nuevas operaciones y puede ejecutar consultas SQL.
- Se puede crear desde:
 - Fuentes de datos externas
 - Resultados de consultas
 - Regular RDDs
 -



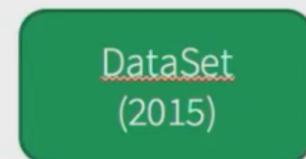
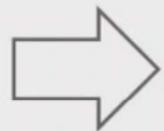
History of Spark APIs



RDD
(2011)



DataFrame
(2013)



DataSet
(2015)

Distribute collection
of JVM objects

Functional Operators (map,
filter, etc.)

Distribute collection
of Row objects

Expression-based operations
and UDFs

Internally rows, externally
JVM objects

Almost the “Best of both
worlds”: type safe + fast

Logical plans and optimizer

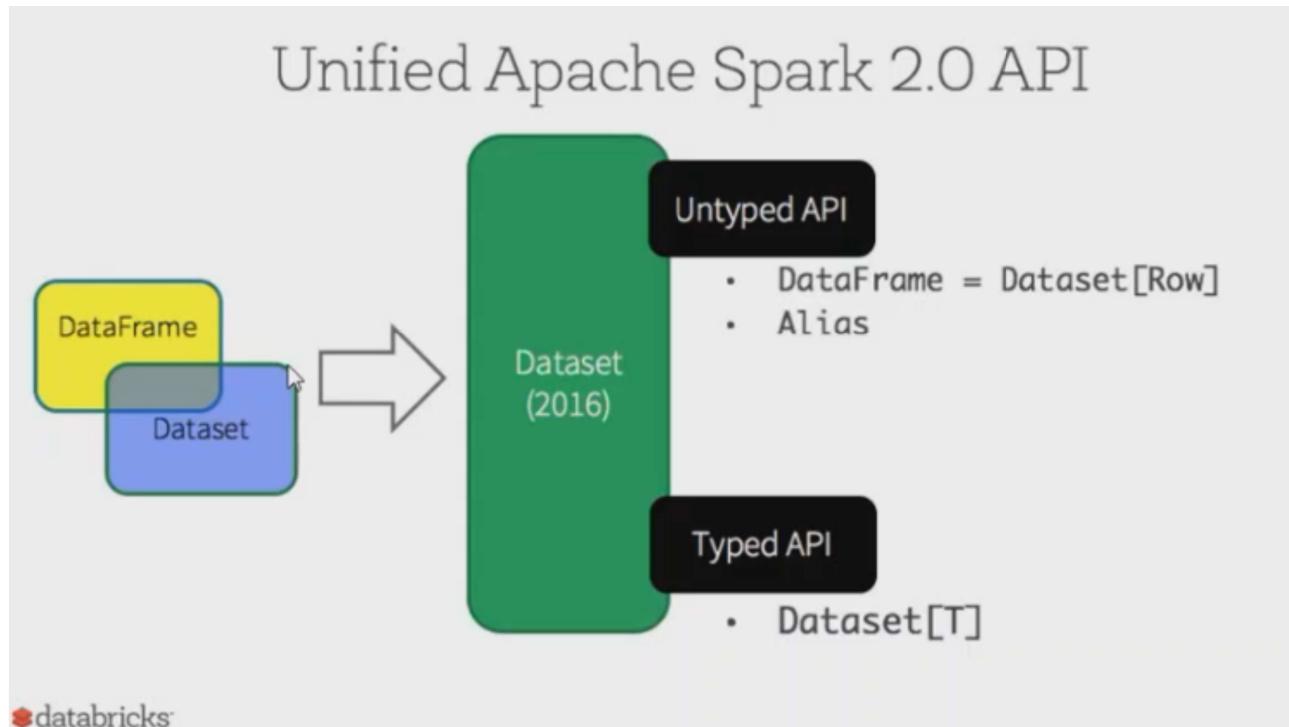
Fast/efficient internal
representations

But slower than DF
Not as good for interactive
analysis, especially Python

 databricks

DataFrames & Datasets

- En Spark 2.0, DataFrames y Datasets se unificaron



DataFrames & Datasets

```
1 fifa_df = spark.read.csv("path-of-file/fifa_players.csv", inferSchema = True, header = True)
2
3 fifa_df.show()
```

RoundID	MatchID	Team Initials	Coach Name	Line-up	Player Name	Position	Event
201	1096	FRA CAUDRON Raoul (FRA)	S	Alex THEPOT	GK	null	
201	1096	MEX LUQUE Juan (MEX)	S	Oscar BONFIGLIO	GK	null	
201	1096	FRA CAUDRON Raoul (FRA)	S	Marcel LANGILLER	null	G40'	
201	1096	MEX LUQUE Juan (MEX)	S	Juan CARRENO	null	G70'	
201	1096	FRA CAUDRON Raoul (FRA)	S	Ernest LIBERATI	null	null	
201	1096	MEX LUQUE Juan (MEX)	S	Rafael GARZA	C	null	
201	1096	FRA CAUDRON Raoul (FRA)	S	Andre MASCHINOT	null	G43' G87'	
201	1096	MEX LUQUE Juan (MEX)	S	Hilario LOPEZ	null	null	
201	1096	FRA CAUDRON Raoul (FRA)	S	Etienne MATTLER	null	null	
201	1096	MEX LUQUE Juan (MEX)	S	Dionisio MEJIA	null	null	
201	1096	FRA CAUDRON Raoul (FRA)	S	Marcel PINEL	null	null	
201	1096	MEX LUQUE Juan (MEX)	S	Felipe ROSAS	null	null	
201	1096	FRA CAUDRON Raoul (FRA)	S	Alex VILLAPLANE	C	null	
201	1096	MEX LUQUE Juan (MEX)	S	Manuel ROSAS	null	null	
201	1096	FRA CAUDRON Raoul (FRA)	S	Lucien LAURENT	null	G19'	
201	1096	MEX LUQUE Juan (MEX)	S	Jose RUIZ	null	null	
201	1096	FRA CAUDRON Raoul (FRA)	S	Marcel CAPELLE	null	null	
201	1096	MEX LUQUE Juan (MEX)	S	Alfredo SANCHEZ	null	null	
201	1096	FRA CAUDRON Raoul (FRA)	S	Augustin CHANTREL	null	null	
201	1096	MEX LUQUE Juan (MEX)	S	Efrain AMEZCUA	null	null	

only showing top 20 rows

DataFrames & Datasets

```
1 fifa_df.printSchema()
```

```
root
|-- RoundID: integer (nullable = true)
|-- MatchID: integer (nullable = true)
|-- Team Initials: string (nullable = true)
|-- Coach Name: string (nullable = true)
|-- Line-up: string (nullable = true)
|-- Player Name: string (nullable = true)
|-- Position: string (nullable = true)
|-- Event: string (nullable = true)
```

DataFrames & Datasets

- DataFrames son Dataset de un **Row object**
- **Row** es un objeto JVM genérico no tipado
- Dataset es una colección fuertemente tipada JVM
- Python no soporta Spark Datasets

```
> case class Person(name: String, age: Int)
   ^
   |
  val personDS = Seq(Person("Max", 33), Person("Adam", 32), Person("Muller", 62)).toDS()
personDS.show()

+---+---+
| name | age |
+---+---+
| Max | 33 |
| Adam | 32 |
|Muller| 62 |
+---+---+
```

DataFrames & Datasets

- DataFrame

```
df.groupBy("dept").avg("age")
```

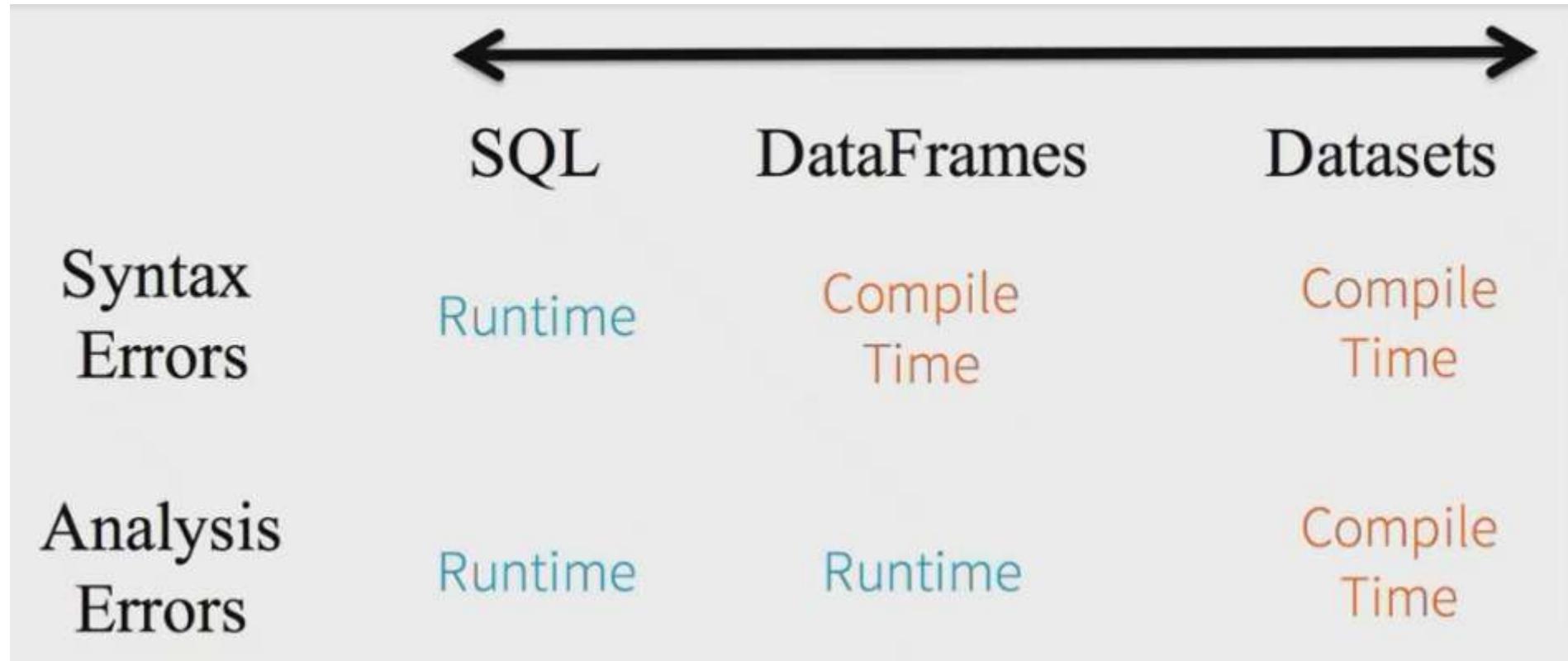
- SQL

```
spark.sql("select dept, avg(age) from data group by dept")
```

- RDD

```
rdd.map { case (dept, age) => dept -> (age, 1) }  
.reduceByKey { case ( (a1, c1), (a2, c2) ) => (a1 + a2, c1 + c2) }  
.map{ case (dept, (age, c)) => dept -> age/ c }
```

DataFrames & Datasets



DataFrames & Datasets

- Abrir “1 – Spark_SQL” en Databricks (fichero de Aules)
 - Ejecuta los pasos 1.1 y 1.2
 - Realiza el ejercicio **Hands on #1**

DataFrame Operadores (org.apache.spark.sql.Column)

Operador	Descripción	Ejemplo df.col ("valor")
==	Igual a	df.col("edad") == 20
!= !=	No igual a Distinto a	df.col("edad") != 20 df.col ("edad" != "NA")
>	Mayor que	df.col("edad") > 18
<	Menor que	df.col("edad") < 65
>= o <=	Mayor o igual que; Menor o igual que	df.col("edad") >= 21; df.col("edad") <= 60

DataFrame UDF

Una User Defined Function (UDF) en Spark es una función definida por el usuario que permite aplicar lógica personalizada a las columnas de un DataFrame o Dataset.

Se usan cuando las funciones nativas de Spark (functions de org.apache.spark.sql.functions) no son suficientes para un caso específico.

¡OJO!

Siempre intenta usar funciones nativas de Spark primero antes de recurrir a UDFs

Puedes usar funciones nativas de Spark (functions.upper(), functions.concat(), etc.).

Trabajas con grandes volúmenes de datos y necesitas rendimiento óptimo.

No necesitas lógica personalizada compleja.

DataFrame UDF

```
val squared = (s: Long) => { s * s }

spark.udf.register("square", squared)

spark.sql("select id, square(id) as id_squared from test")
```

- Usando UDFs con DataFrames

```
import org.apache.spark.sql.functions.{col, udf}
val squared = udf((s: Long) => s * s)
df.select(squared(col("id")) as "id_squared")
```

Ejercicio

- Abrir “1 – Spark_SQL” en Databricks (fichero de Aules)
 - Realiza el ejercicio **Hands on #2 hasta #6**

DataFrames – SQL de ficheros

- En lugar de utilizar la API de lectura para cargar un archivo en DataFrame y consultararlo, también puede consultar ese archivo directamente con SQL

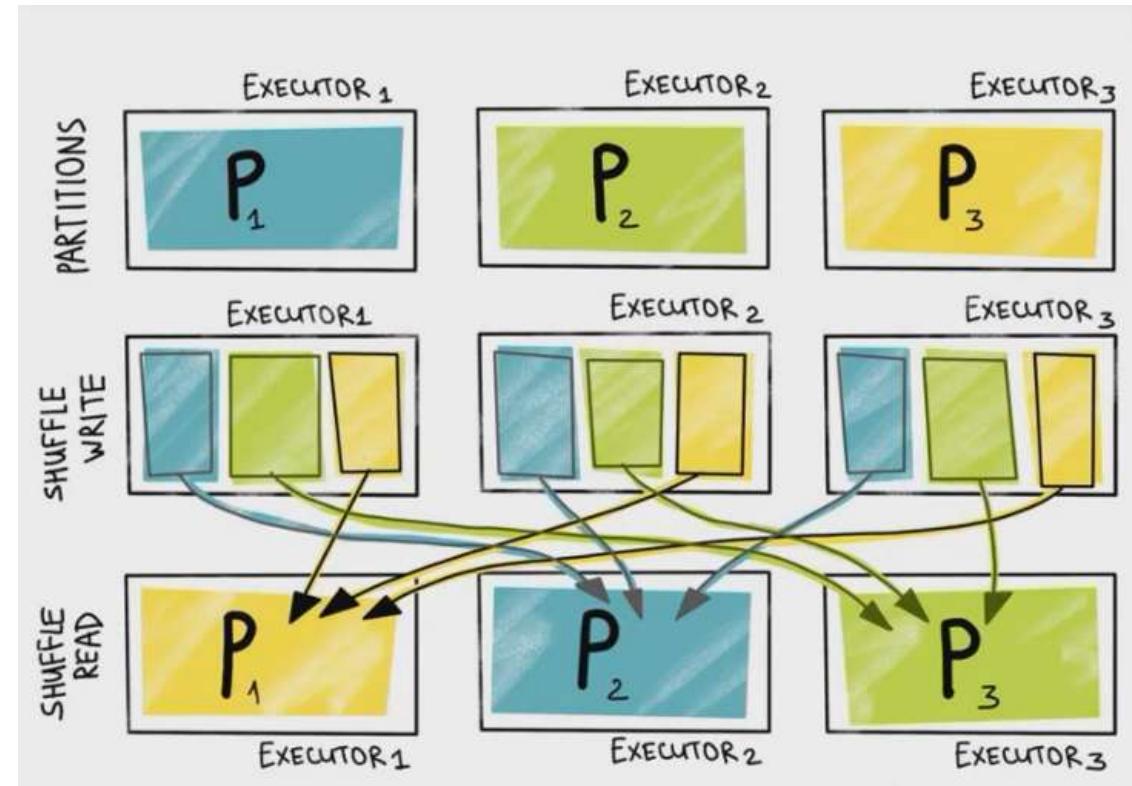
```
df = spark.sql ("SELECT * FROM parquet.`./resources/users.parquet`")
```

```
df = spark.sql ("SELECT * FROM json.`./resources/users.json`")
```

```
df = spark.sql ("SELECT * FROM orc.`./resources/users.orc`")
```

DataFrames Partitioning

- El particionado de datos es **crítico** para el procesamiento de datos, especialmente para grandes volúmenes de datos.
- Spark **asigna una tarea para cada partición**
 - Si tenemos pocas particiones, no se utiliza todos los executors.
 - Demasiadas particiones → Overhead



DataFrames Partitioning - Overhead

Cada partición en Spark se convierte en una tarea (task) genera sobrecarga en el Job Scheduler y un mayor consumo de memoria (Driver y Executors).

Cada tarea necesita recursos en memoria. Si hay demasiadas, el Driver y los Executors pueden saturarse con metadatos de tareas, provocando GC (Garbage Collection) excesivo.

Si las particiones son demasiado pequeñas, el tiempo de ejecución de cada tarea es menor que el tiempo que Spark necesita para iniciarla, provocando que pase más tiempo gestionando tareas que ejecutándolas realmente.

Si hay muchas particiones pequeñas, el Shuffle (intercambio de datos entre nodos) aumenta, lo que genera una gran cantidad de transferencia de datos en la red y ralentiza la ejecución.

Repartition vs Coalesce

¿Se puede optimizar el número de particiones?

- De 2 a 4 veces el número de núcleos disponibles en el clúster. Por defecto, spark.sql.shuffle.partitions = 200 para grandes volúmenes de datos.

Repartionar los datos

Repartition nos permite incrementar o decrementar el número de particiones. Tiene Shuflle operation.

```
val new_df = df.repartition(100)  
val new_df = df.repartition(100, $"id")
```

Reducir el número de particiones con Coalesce. Evita shuffle.

```
val new_df = df.coalesce(10) .-> reduce a n particiones.
```

Tener muchas particiones genera overhead porque incrementa la gestión de tareas, el consumo de memoria y la latencia en la ejecución.

Repartition vs Coalesce

Repartitioning

19M repartition/part-00000
19M repartition/part-00001
19M repartition/part-00002
19M repartition/part-00003
19M repartition/part-00004
19M repartition/part-00005
19M repartition/part-00006
19M repartition/part-00007
19M repartition/part-00008
19M repartition/part-00009

Coalesce

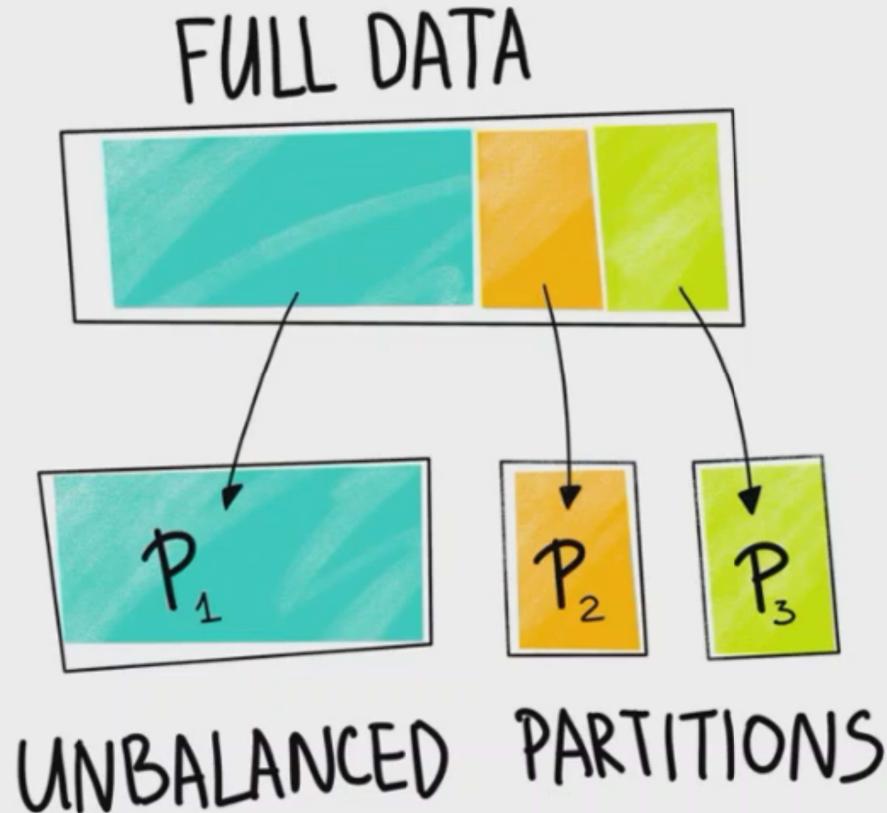
33M coalesce/part-00000
29M coalesce/part-00001
30M coalesce/part-00002
31M coalesce/part-00003
32M coalesce/part-00004
33M coalesce/part-00005

Data Skew

Data Skew ocurre cuando los datos están desbalanceados entre las particiones de Spark que implica que algunas particiones contienen muchos más datos que otras particiones, causando problemas como:

- **Carga de trabajo desbalanceado.** Un ejemplo executors que tardan más en completar el trabajo haciendo que Spark deba esperar a que terminar para poder continuar.
- **Exceso Garbage Collection (GC)** pudiendo provocar posibles errores de memoria.
- **Operaciones ineficientes como join** porque al tener particiones desbalanceadas una partición termina con demasiada carga mientras que otras están casi vacías.

Data Skew



@luminousmen.com

Data Skew ¿Qué hacer?

1. Reparticiona los datos por una clave más distribuida (si tienes alguna).

2. Broadcast Join

- Distribuye (broadcast) un dataframe pequeña a todos los nodos del clúster, evitando el costoso shuffle (intercambio de datos entre nodos) durante un join.
- Especialmente útil cuando tenemos un dataframe pequeño y un dataframe grande y queremos evitar el desbalanceo de datos y overhead de procesamiento.

¿Por qué?

- Evita shuffle, porque envía a todos los nodos el dataframe, eliminando la necesidad de mover el dataframe grande.
- Como los datos están en cada nodo, el join se realiza rápidamente.
- Reduce el uso de memoria lo que evita problemas de memoria y Garbage Collection.

Data Skew ¿Qué hacer?

3. Usar una **random key** (clave aleatoria) puede ayudar a mejorar la distribución de los datos y reducir el problema de Data Skew, especialmente en operaciones como join o group by.

Cuando Spark ejecuta estas operaciones, usa la clave de agrupamiento o join para distribuir los datos en particiones. Si algunas claves tienen muchos más registros que otras, estas particiones producen desbalanceo. El salting introduce una clave aleatoria para distribuir los datos de manera más uniforme entre las particiones.

Data Skew. Conclusiones

El Data Skew puede hacer que algunos nodos trabajen demasiado mientras otros están inactivos, ralentizado todo el proceso.

Las soluciones como **repartition**, **salting** y **broadcast join** pueden ayudar a mejorar el rendimiento y balancear la carga de trabajo.

SparkSession

- **SparkSession** introducido en Spark 2.0
- Todos los contextos unificados en uno solo
- Teniendo acceso a **SparkSession**, automáticamente tenemos acceso a **SparkContext**.
- SparkSession es ahora el nuevo punto de entrada.
- Podemos comenzar a trabajar con **DataFrame** y **DataSet** teniendo acceso a **SparkSession**

```
val spark = SparkSession  
  .builder  
  .enableHiveSupport()  
  .config("spark.master", "local[*]")  
  .appName("Simple Application")  
  .getOrCreate()
```

```
val persons = spark  
  .read  
  .json(filePath)
```

Spark Application

```
val spark = SparkSession
  .builder
  .enableHiveSupport()
  .config("spark.master", "local[*]")
  .appName("Simple Application")
  .getOrCreate()

val filePath = "person.json"
val persons = spark
  .read
  .json(filePath)

val personsByName = persons
  .groupBy(col("name"))
  .count()

personsByName.show()
```

name	count
Michael	2
John	1

Hive Metastore

▪ *Hive Metastore*

Catálogo centralizado que almacena metadatos sobre tablas y bases de datos. Actúa como una base de datos para la información de los esquemas, ubicaciones de archivos y otras propiedades en formatos como Parquet, ORC o CSV.

- Spark consulta Hive Metastore cada vez que accede a una tabla, para conocer su estructura y ubicación.
- Los metadatos se almacenan en una base de datos.
- Spark lee y escribe los datos en HDFS o cualquier almacenamiento compatible, pero usa el Metastore para saber cómo están organizados.

Hive Metastore

- Configuración de Spark con Hive

spark-shell --conf spark.sql.catalogImplementation=hive

- Iniciar Spark Sesión con soporte Hive

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

```
val spark = SparkSession.builder()
.appName("Spark_Hive_Empieados")
.enableHiveSupport() // Habilita soporte para Hive Metastore
.getOrCreate()
```

Hive Metastore

- Nueva una base de datos en Hive

```
spark.sql("CREATE DATABASE IF NOT EXISTS empresa_db")
```

```
spark.sql("USE empresa_db")
```

- Nueva tabla de empleados y los datos se almacenan en HDFS.

```
spark.sql(" CREATE TABLE IF NOT EXISTS empleados (id INT, nombre STRING, edad INT,  
departamento STRING, salario DOUBLE )  
STORED AS PARQUET")
```

*PARQUET es un formato de almacenamiento columnar optimizado para consultas rápidas con grandes volúmenes de datos y almacenamiento eficiente en sistemas distribuidos como Spark.

SparkContext – SQL Context - HiveContext

SparkContext

- The driver program use the SparkContext to connect and communicate with the cluster (YARN, Mesos)
- Using SparkContext you can actually get access to SQLContext and HiveContext
- Setting configuration parameters

```
val sparkConf = new SparkConf().setAppName("Example").setMaster("yarn")
val sc = new SparkContext(sparkConf)
```

SQLContext

- SQLContext is your gateway to SparkSQL

```
// sc is an existing SparkContext. (Deprecated)
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
```

HiveContext

- HiveContext is your gateway to Hive
- HiveContext extends SQLContext, including Hive functionalities

```
// sc is an existing SparkContext. (Deprecated)
val sqlContext = new org.apache.spark.sql.hive.HiveContext(sc)
```

Saved to Persistent Tables

- DataFrames puede ser guardados como tablas persistentes usando el comando **saveAsTable**
 - A diferencia del comando **registerTempTable**, **saveAsTable** materializará el contenido del marco de datos y creará un puntero a los datos en el *HiveMetastore*.
 - Persisten Tables todavía estarán después Spark program finalice.
 - Por defecto saveAsTable creará un “managed table”, lo que significa que la ubicación de los datos estará controlada por el metastore.
 - Los datos de las tablas gestionadas también se borrarán automáticamente cuando se elimine una tabla.

Window Functions

- SparkSQL admite dos tipos de funciones que pueden utilizarse para calcular un único valor de retorno:
 - Como “substr”, “round”, etc.
 - Funciones agregadas como SUM, MAX, etc.
- **Window Function** permiten realizar cálculos sobre un conjunto de filas relacionadas con la fila actual dentro de una partición de dato.
- **Window Function** son más potentes que otras funciones, permitiendo realizar tareas de procesamientos de datos complicados.
- Se usan comúnmente para cálculos como rankings, acumulados y funciones de agregación en un contexto específico.

Window Functions. Ejemplo 1

productRevenue		
product	category	revenue
Thin	Cell phone	6000
Normal	Tablet	1500
Mini	Tablet	5500
Ultra thin	Cell phone	5000
Very thin	Cell phone	6000
Big	Tablet	2500
Bendable	Cell phone	3000
Foldable	Cell phone	3000
Pro	Tablet	4500
Pro2	Tablet	6500

¿Cuáles son los productos más vendidos y los segundos más vendidos de cada categoría?

```
SELECT
    product,
    category,
    revenue
FROM (
    SELECT
        product,
        category,
        revenue,
        dense_rank() OVER (PARTITION BY category ORDER BY revenue DESC) as rank
    FROM productRevenue) tmp
WHERE
    rank <= 2
```

Window Functions. Ejemplo 1

productRevenue		
product	category	revenue
Thin	Cell phone	6000
Normal	Tablet	1500
Mini	Tablet	5500
Ultra thin	Cell phone	5000
Very thin	Cell phone	6000
Big	Tablet	2500
Bendable	Cell phone	3000
Foldable	Cell phone	3000
Pro	Tablet	4500
Pro2	Tablet	6500



Product	category	revenue	rank
Thin	Cell phone	6000	1
Very thin	Cell phone	6000	1
Ultra thin	Cell phone	5000	2
Foldable	Cell phone	3000	3
Bendable	Cell phone	5000	3

Window Functions. Ejemplo 1

productRevenue		
product	category	revenue
Thin	Cell phone	6000
Normal	Tablet	1500
Mini	Tablet	5500
Ultra thin	Cell phone	5000
Very thin	Cell phone	6000
Big	Tablet	2500
Bendable	Cell phone	3000
Foldable	Cell phone	3000
Pro	Tablet	4500
Pro2	Tablet	6500



Product	category	revenue	rank
Pro2	Tablet	6500	1
Mini	Tablet	5500	2
Pro	Tablet	4500	3
Big	Tablet	2500	4
Normal	Tablet	1500	5

Window Functions. Ejemplo 1

productRevenue		
product	category	revenue
Thin	Cell phone	6000
Normal	Tablet	1500
Mini	Tablet	5500
Ultra thin	Cell phone	5000
Very thin	Cell phone	6000
Big	Tablet	2500
Bendable	Cell phone	3000
Foldable	Cell phone	3000
Pro	Tablet	4500
Pro2	Tablet	6500



product	category	revenue
Pro2	Tablet	6500
Mini	Tablet	5500
Thin	Cell Phone	6000
Very thin	Cell Phone	6000
Ultra thin	Cell Phone	5500

Window Functions. Ranking functions. SQL – DataFrame API

- **Rank**: es una window function que asigna un número de ranking a cada fila dentro de una partición, basado en un orden específico.
- **dense_rank** es una window function que asigna un número de ranking a cada fila dentro de una partición en un orden específico. La diferencia con Rank es que no deja huecos de enumeración y si hay valores repetidos, comparten el mismo rango.
- **percent_rank** es una window function que calcula el porcentaje de rango de cada fila dentro de una partición.
- **ntile** es una window function que divide las filas de una particion en n grupos aprox. Iguales asignado a cada fila un número de grupo (del 1 al n).
- **row_number** es una window function que asigna un número secuencial único a cada fila dentro de una partición ordenado por un criterio.

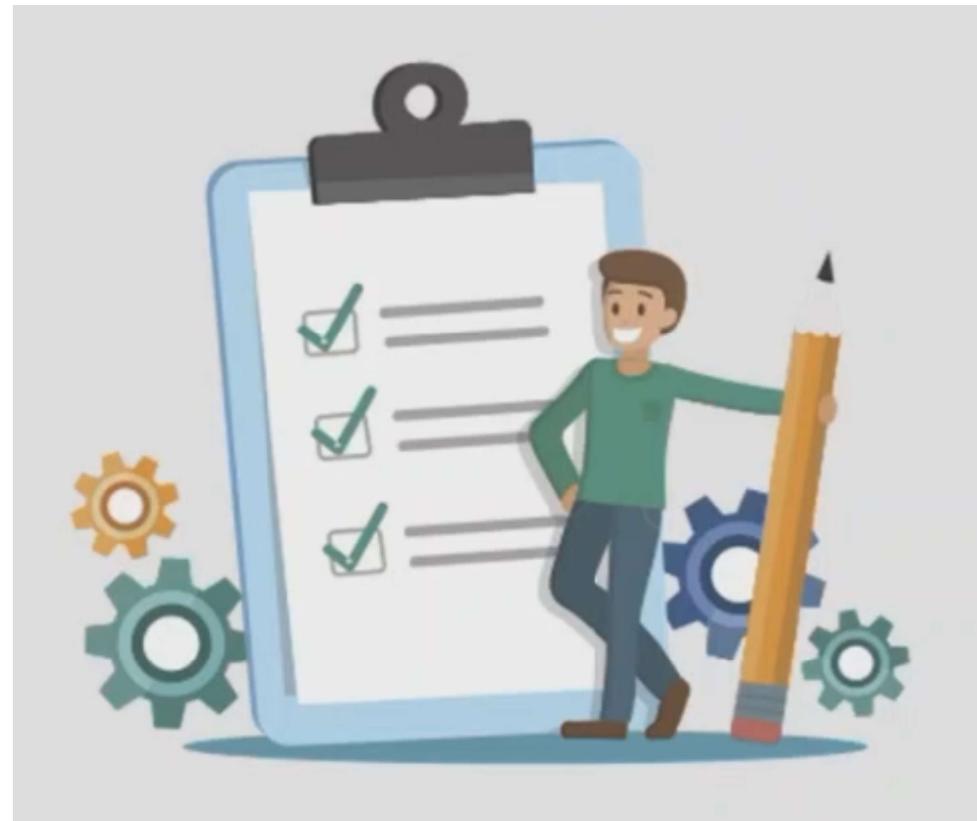
Window Functions. Analytic Functions. SQL – DataFrame API

- **cume_dist**: es una window function que calcula la distribución acumulativa de cada fila dentro de una partición. Es un concepto fundamental en estadística y se utiliza para entender cómo se distribuyen los valores de una variable en un conjunto de datos.
- **first_value** es una window function que devuelve el primer valor dentro de una partición ordenada según un criterio específico.
- **last_value** es una window function que devuelve el último valor dentro de una partición, basado en un criterio de orden específico.
- **lag** es una window function que permite acceder al valor de una columna en una fila anterior dentro de la partición definida. Es útil cuando necesitas comparar un valor con el de la fila anterior en una secuencia.
- **lead** es una window function que permite acceder al valor de una columna en una fila siguiente dentro de la partición definida, en lugar de acceder a una fila anterior, como hace lag().

Ejercicios en Databricks

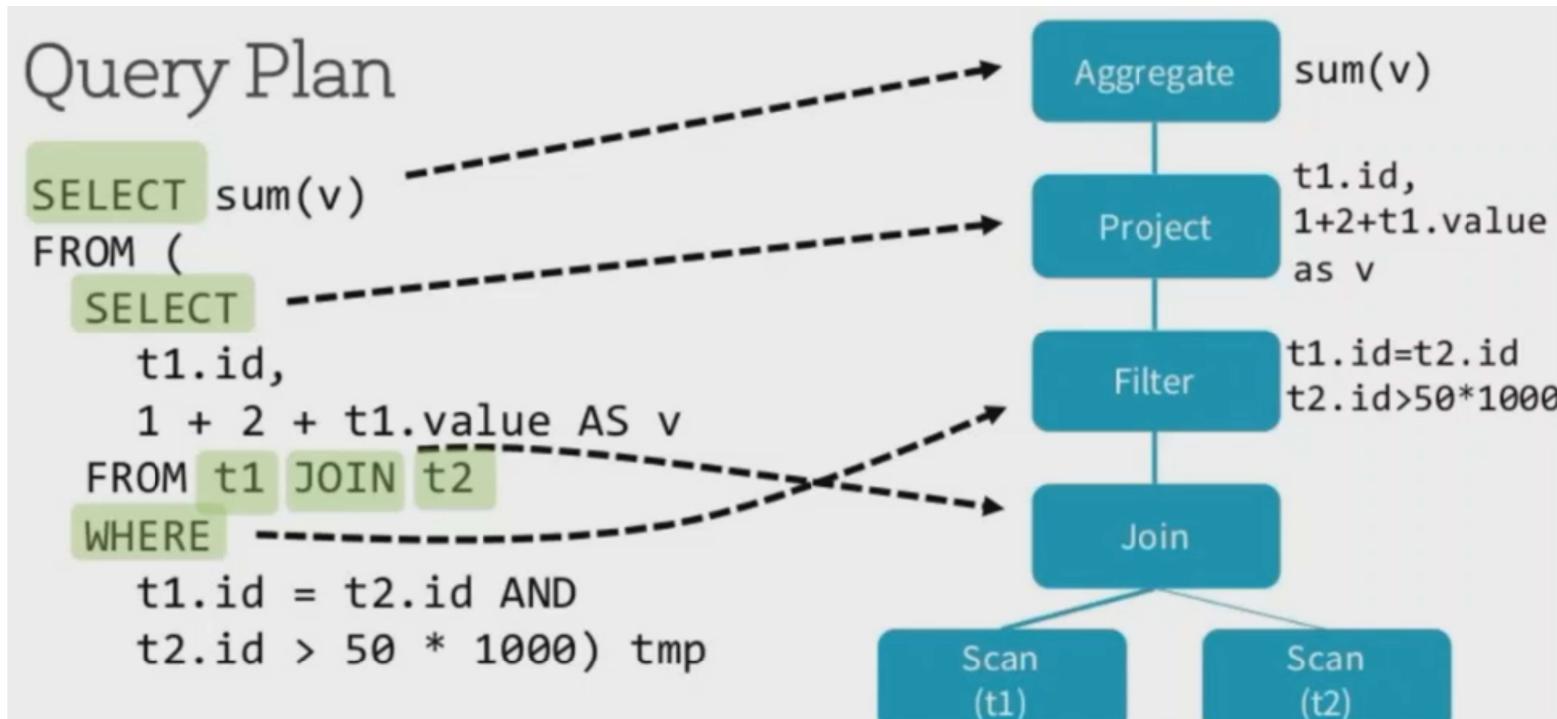
Abrir “2 – Spark_SQL_Advanced” en Databricks

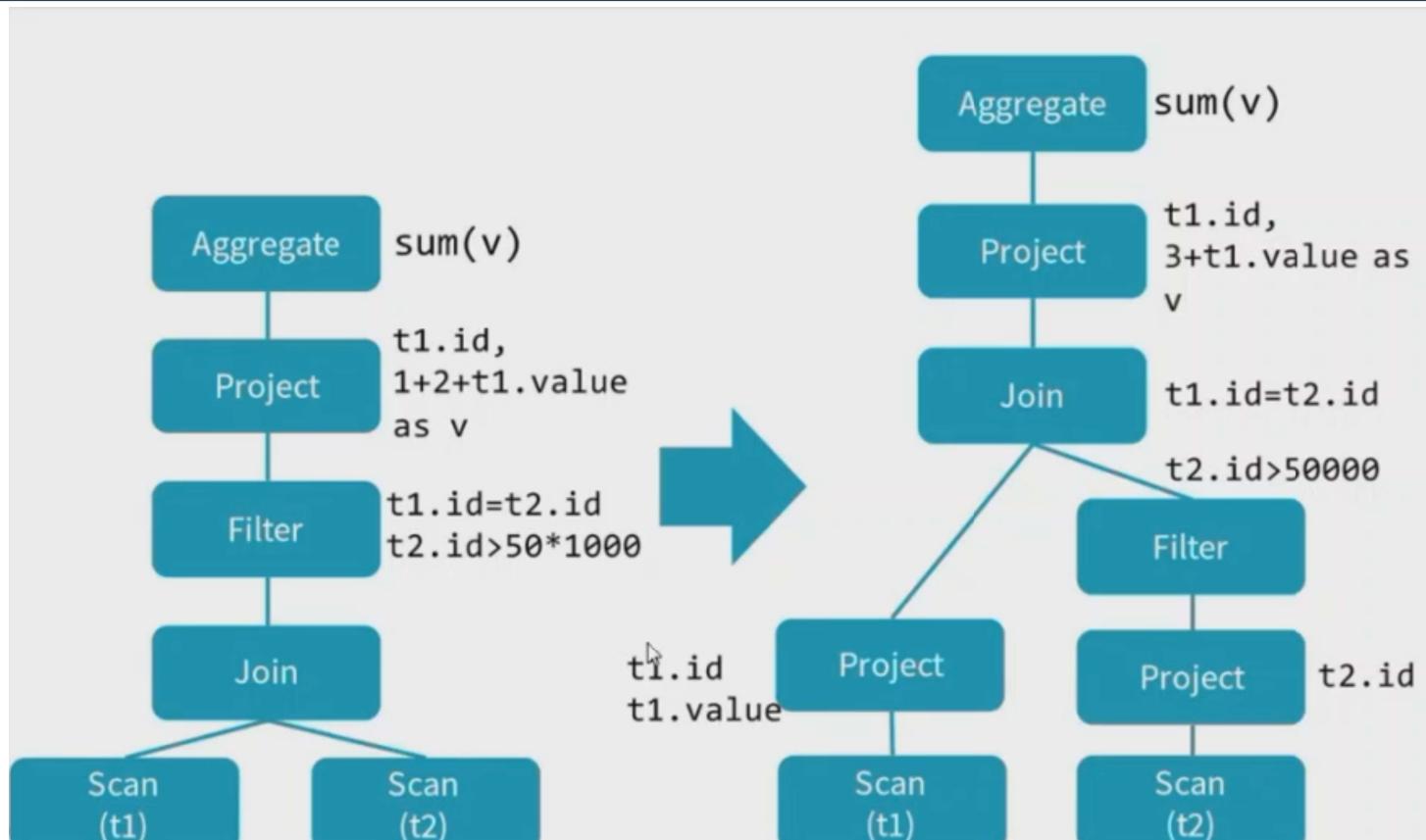
- Ejecutar el 1.1 section
- Realizar el Hands on #1 y #2



Catalyst

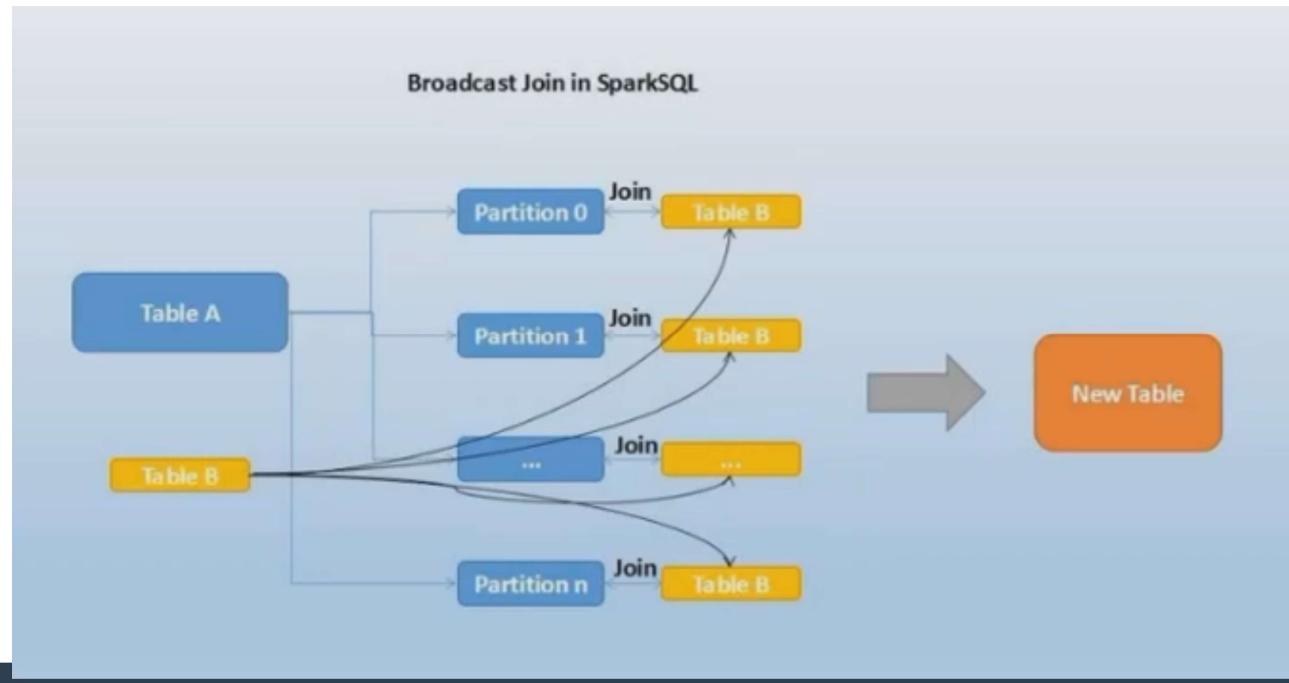
Catalyst averigua el plan de ejecución más eficiente





Joins en Spark SQL. Broadcast Hash Join

- Tabla grande/ mediana – Tabla pequeña
- Tabla pequeña debe ser replicada para cada executor
- La idea principal es evitar shuffle.
- Las tablas pequeñas de menos de 10 Megas se realiza el Broadcast automáticamente.
spark.sql.autoBroadcastJoinThreshold (10M)



Joins en Spark SQL. Shuffled Hash Join

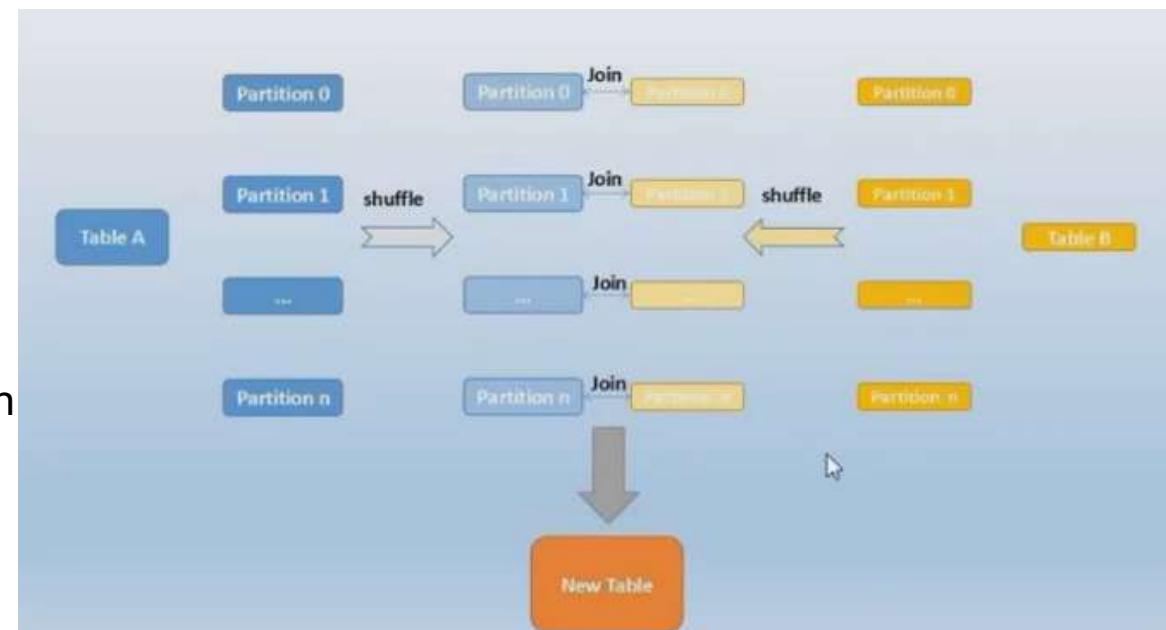
El Shuffled Hash Join en Apache Spark es un tipo de Hash Join que se usa cuando ninguna de las tablas es lo suficientemente pequeña como para aplicar un Broadcast Hash Join, pero sí es posible hacer un Hash Join dividiendo los datos en particiones.

- **Shuffle:** Spark reparticiona ambas tablas en el clúster con la misma clave de join terminan en la misma partición.

- **Construcción de un Hash Table:**

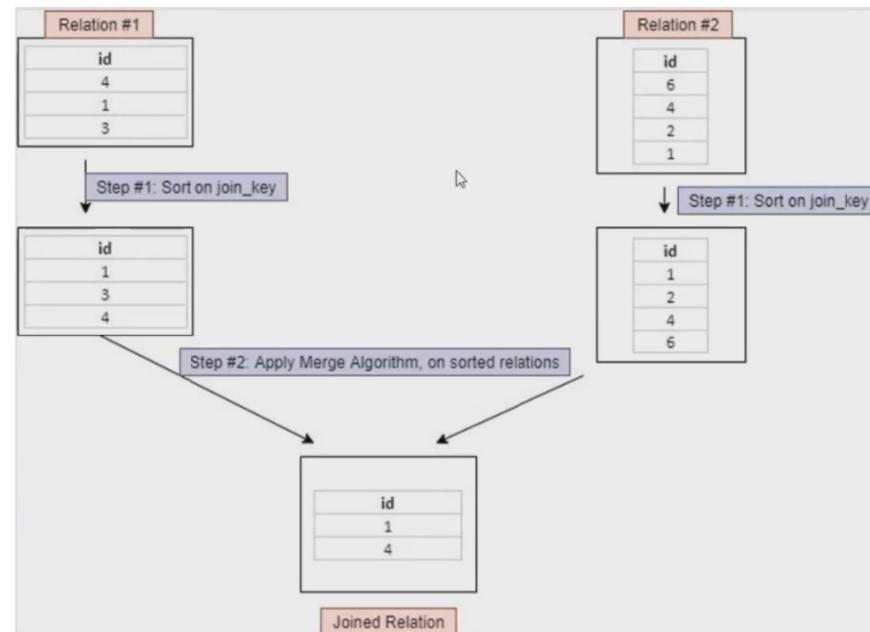
Para cada partición, Spark elige una de las tablas y construye una tabla hash en memoria.

- **Búsqueda en la tabla hash:** Luego, Spark escanea la otra tabla en la misma partición y busca coincidencias en la tabla hash. Se devuelven los resultados combinados.



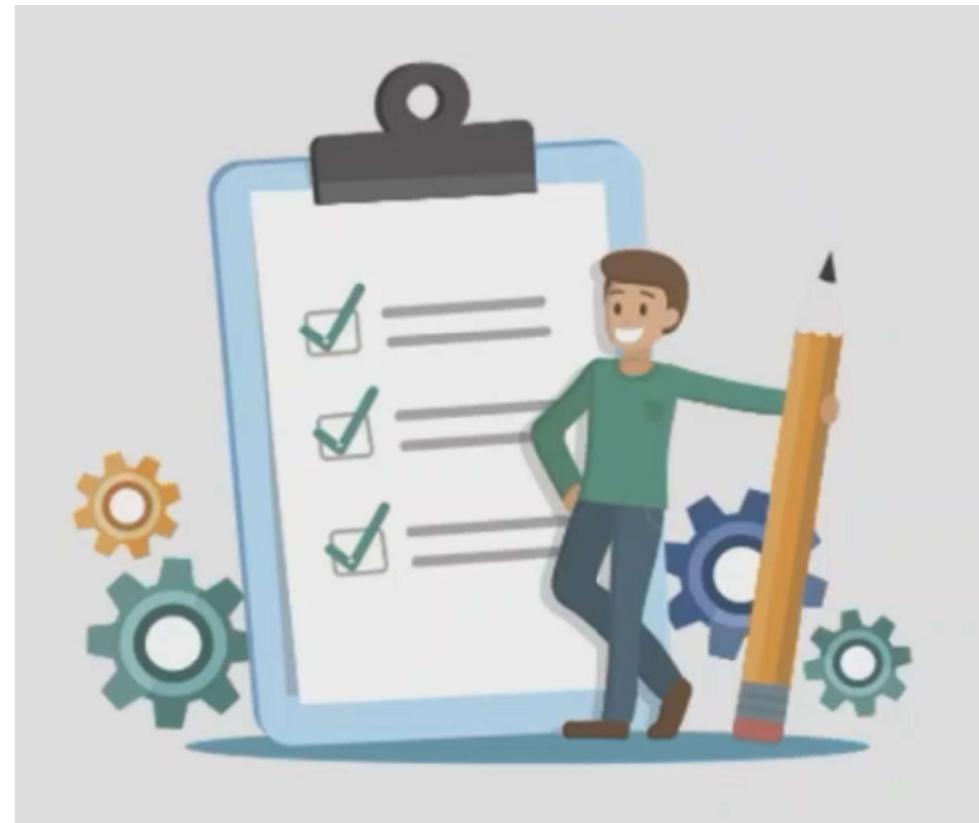
Joins en Spark SQL. Sort Merge Join

- Cuando las dos tablas son muy grandes entonces un **Sort Merge Join**.
- Los datos están ordenados antes de hacer el join
- Si lo comparamos con shuffle hash join, este Join podría necesitar guardar en disco.
- `spark.sql.join.preferSortMergeJoin` la propiedad por defecto es *True*



Ejercicios en Databricks

Abrir “2 – Spark_SQL_Advanced” en Databricks
- Realizar el Hands on #3



AQE: Adaptive Query Execution

Spark elige la estrategia de join basándose en estadísticas estimadas antes de ejecutar la consulta.

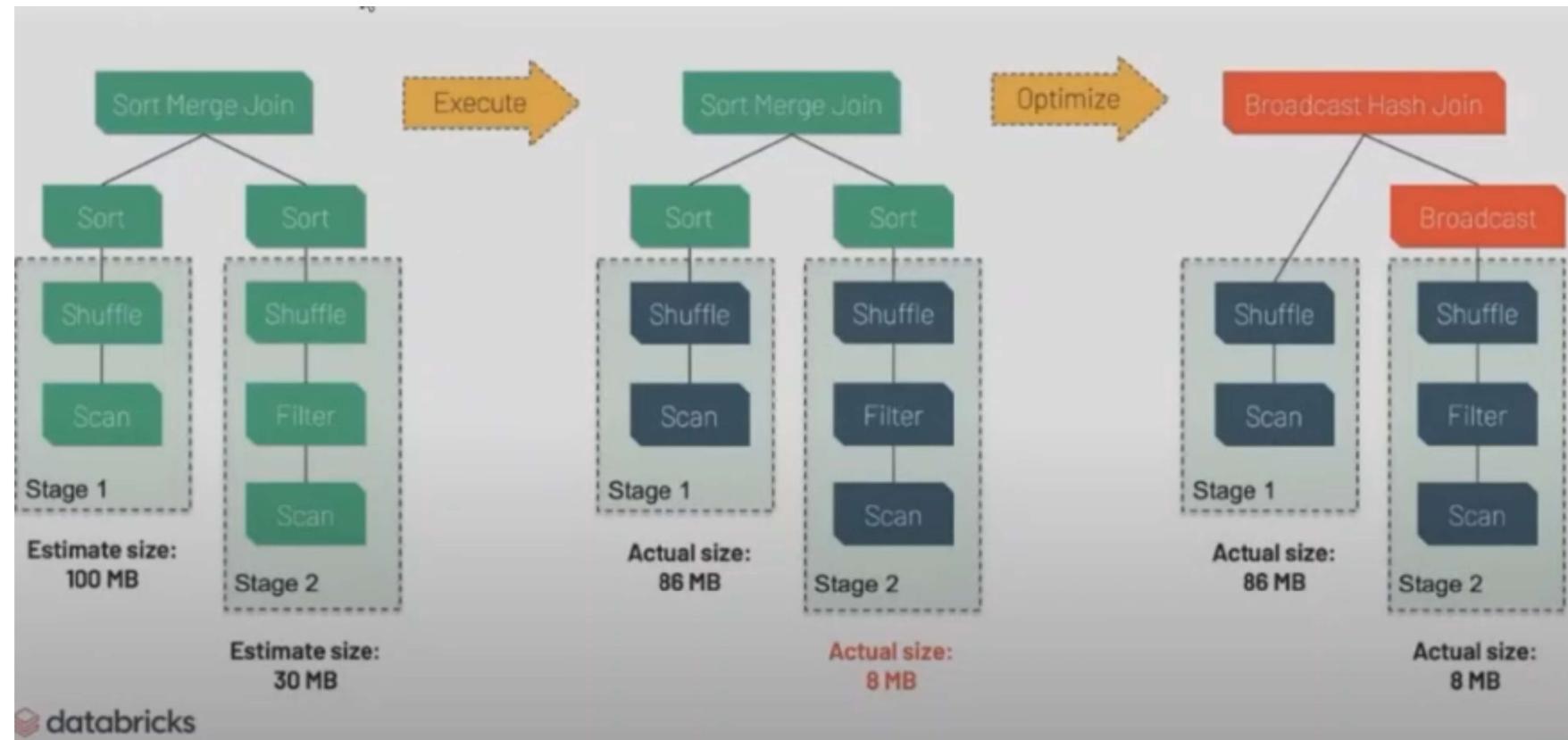
Con AQE, Spark puede cambiar dinámicamente la estrategia de join en tiempo de ejecución, por ejemplo:

- Convertir un **SortMergeJoin** en un **BroadcastHashJoin** si detecta que una de las tablas es lo suficientemente pequeña para enviarla a todos los nodos.
- Esto reduce el coste de shuffle y mejora la velocidad de ejecución.

if spark.sql.adaptive.localShuffleReader.enabled is true

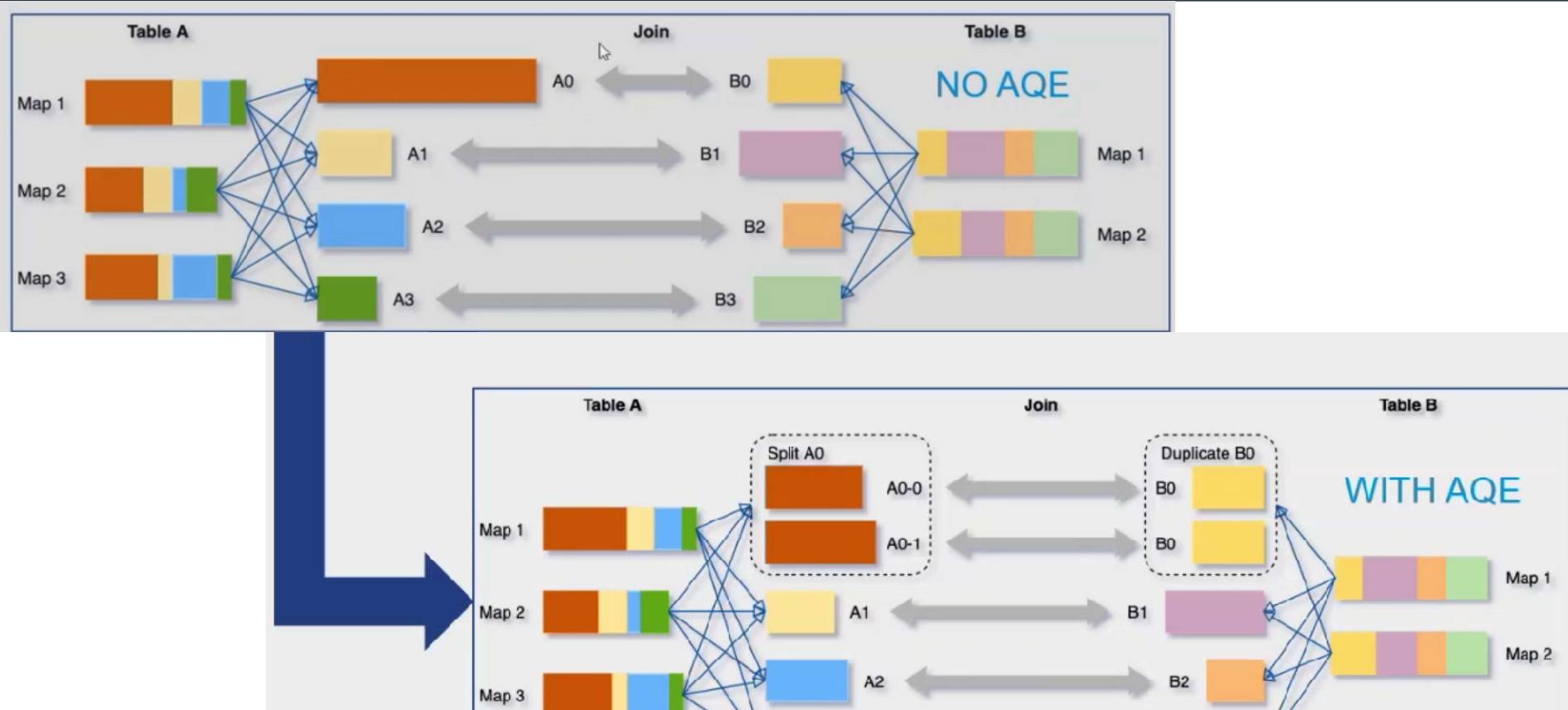
AQE: Adaptive Query Execution

spark.sql.adaptive.enabled



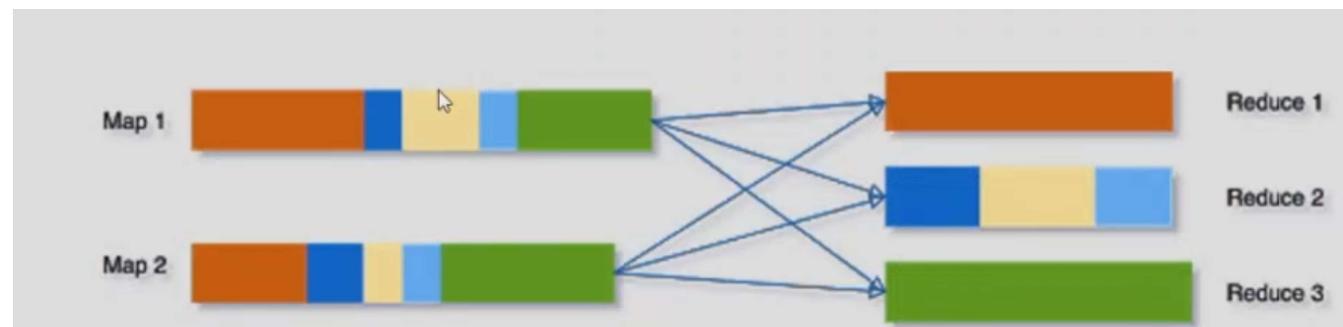
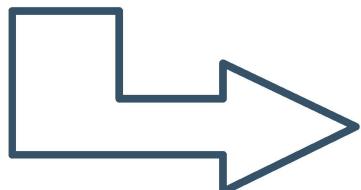
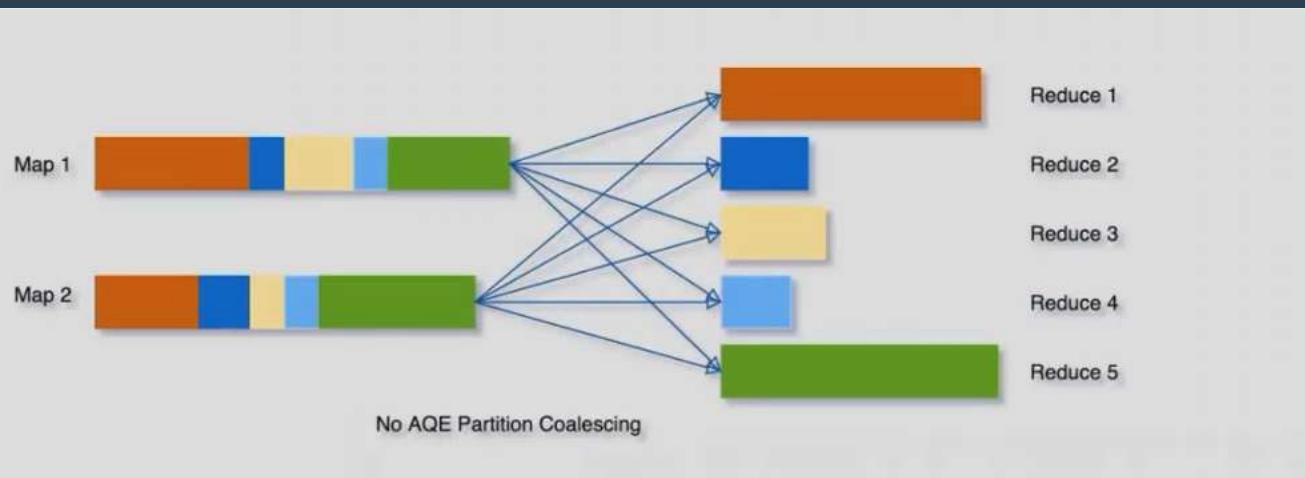
AQE: Data Skew Optimization

spark.sql.adaptive.skewJoin.enabled



AQE: Colaese del número shuffle partitions

spark.sql.adaptive.coalescePartitions.enabled



DataFrame - Caching

- **Cache** : cache en memoria solo

- **Persist**: se usa para almacenar con nivel definido por el usuario

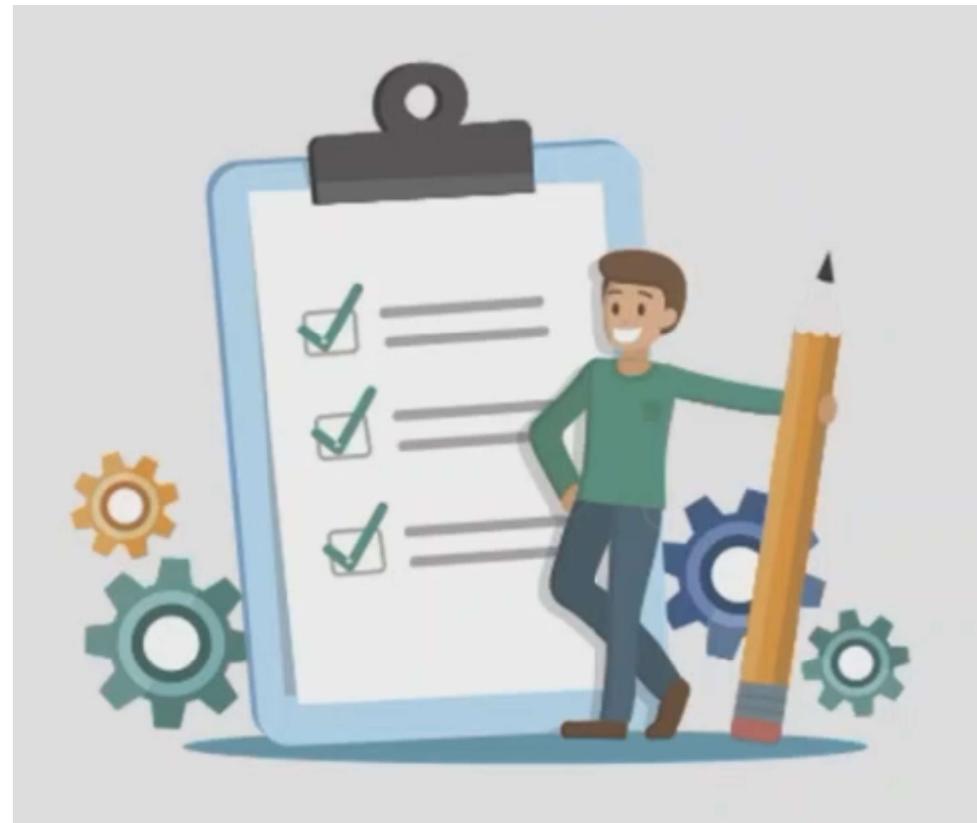


- **UnPersist**: borra de Spark el DataFrame de cache

Level	Space used	CPU time	In memory	On disk	Comments
MEMORY_ONLY	High	Low	Y	N	
MEMORY_ONLY_SER	Low	High	Y	N	
MEMORY_AND_DISK	High	Medium	Some	Some	Spills to disk if there is too much data to fit in memory.
MEMORY_AND_DISK_SER	Low	High	Some	Some	Spills to disk if there is too much data to fit in memory. Stores serialized representation in memory.
DISK_ONLY	Low	High	N	Y	

Ejercicios en Databricks

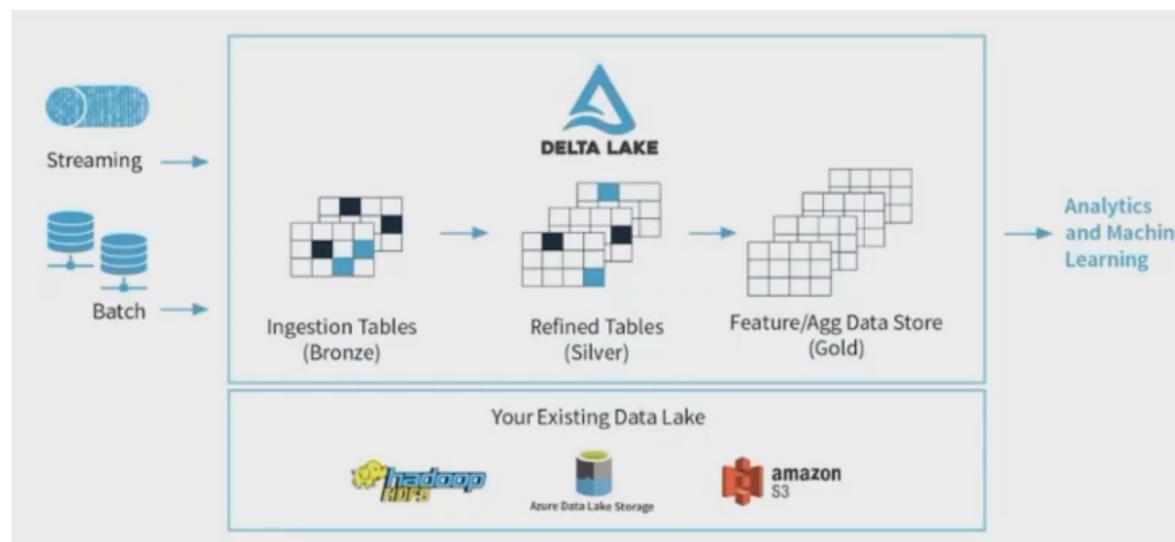
Abrir “2 – Spark_SQL_Advanced” en Databricks
- Realizar el Hands on #3.1



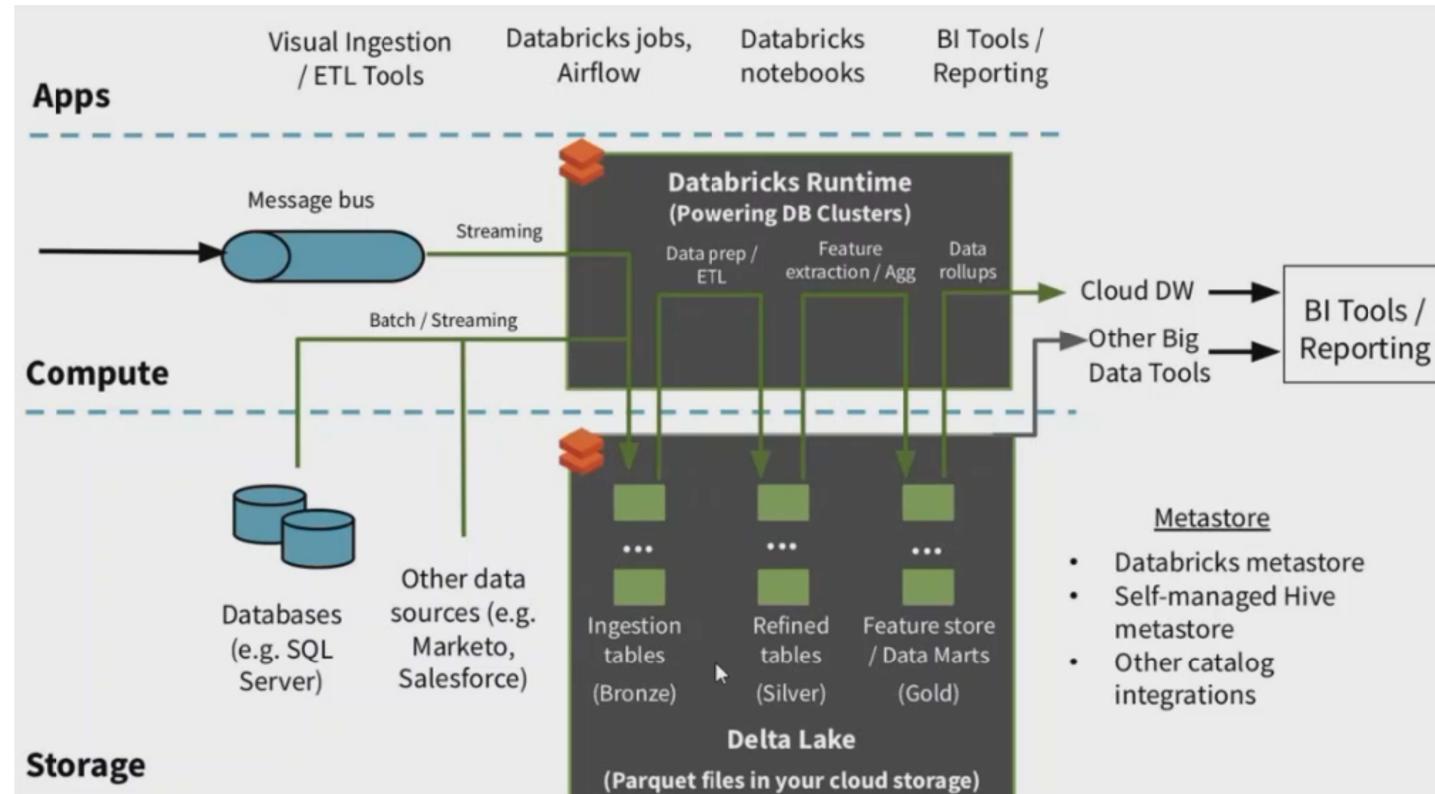
¿Qué es Delta Lake?

Delta Lake es una capa de almacenamiento de código abierto que funciona sobre **Apache Spark**, ofreciendo transacciones ACID, control de versiones y un rendimiento optimizado para **Data Lakes** en entornos de Big Data. Desarrollado por Databricks, es una tecnología clave en las arquitecturas Lakehouse.

*ACID: Atomicidad, Consistencia, Aislamiento, Durabilidad.



Arquitectura



ACID Transaction

Delta Lake proporciona **ACID Transaction**

ACID: Atomicidad, Consistencia, Aislamiento, Durabilidad.

Significa una unidad de trabajo (es todo o nada), funciona de forma predecible, múltiples usuarios pueden trabajar concurrentemente sin interferirse unos con otros y tienes la garantía que la transaccionalidad sobrevivirá al fallo del sistema.

Delta Lake Transaction Log es un log ordenado de todas las transacciones

Spark comprueba el transaction log para saber qué transacciones se han hecho sobre qué tabla actualizando los datos.

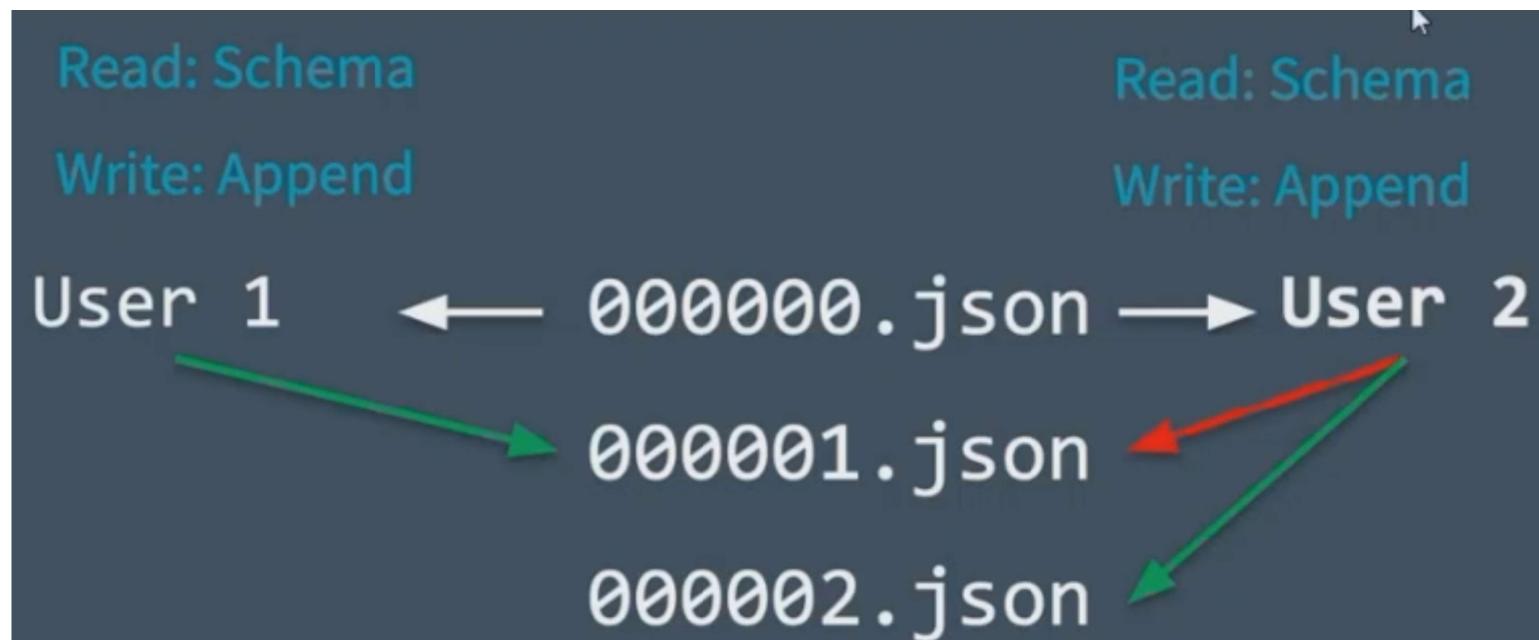
Atomicidad: si no se ha grabado en el transaction log, nunca ha ocurrido.

Todos los cambios en la base de datos son atómicos, commits y ordenados en el transaction Log.

ACID Transaction

Salvando conflictos de forma optimista:

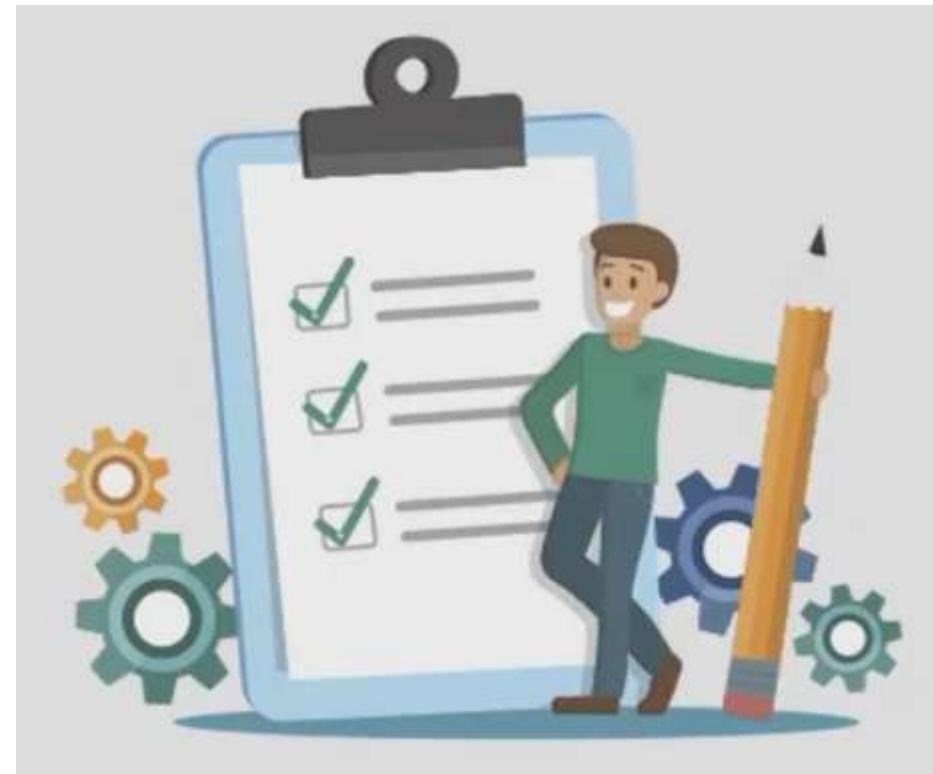
Asume que las transacciones (cambios) realizados en una tabla por un usuario pueden estar completas sin tener conflictos con otros usuarios.



Ejercicios

Abrir 3 – Spark_Delta en Databricks

- Ejecuta los pasos 1.1 y 1.2
- Realiza hands on #1

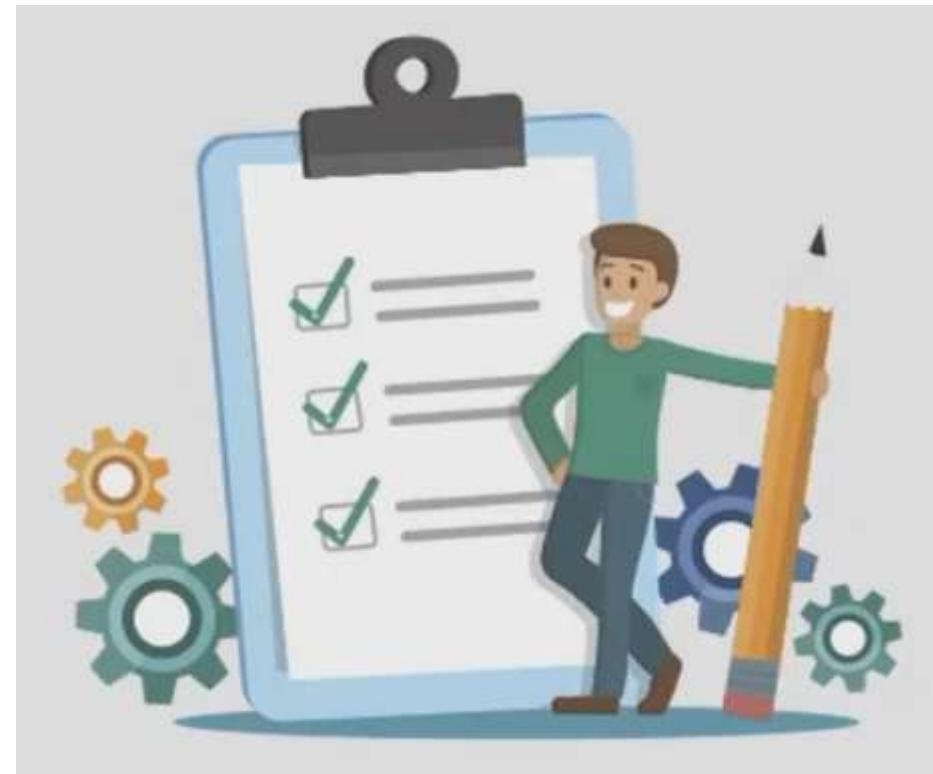


Time Travel

- Modificación de los datos:
 - Auditar los cambios de los datos
 - Realizar reports
 - Rollbacks
- Todas las operaciones son automáticamente versionadas en Delta Lake
- Podemos acceder a diferentes versiones de los datos en dos momentos diferentes:
 - Usando timestamp
 - Usando números de versión

Ejercicios

Abrir 3 – Spark_Delta en Databricks
- Ejecuta los pasos 2.1 y 2.2



Schema Enforcement & Evolution

Schema enforcement (Schema validation) previene que los usuarios puedan accidentalmente romper las tablas con errores o garbage data.

- Rechaza los writes de una tabla que no match con la tabla schema
- Schema validation ocurre on write
- Si el schema no es compatible, Delta Lake cancela la transacción:
 - No puede contener ninguna columna adicional
 - No puede tener tipos de datos diferentes que los tipos de datos de las columnas
 - No puede contener nombres de columnas diferentes
- Lanza una excepción

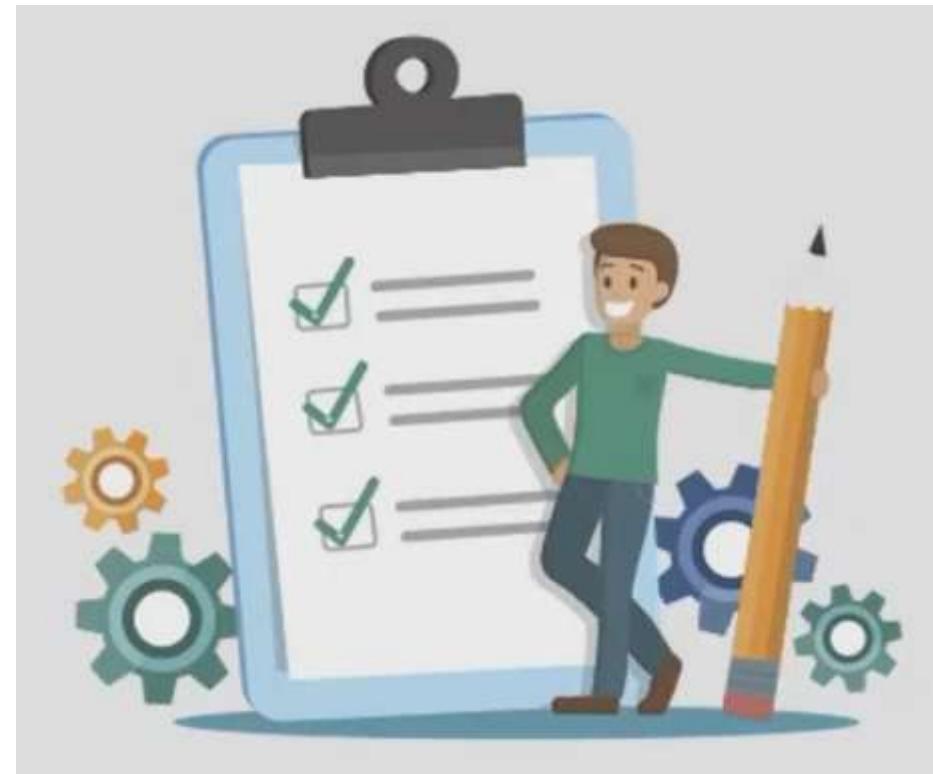
Schema Enforcement & Evolution

- Schema evolution es una forma de evolucionar el schema y añadir nuevas columnas automáticamente.
- Permite adecuar todas las tablas a todos los cambios que existen en negocio.
- Más utilizado para
 - Append
 - Overwrite
- Usa “.option(‘mergeSchema’,’true’) o “.option(‘overwriteSchema’,’true’) y esto nos permite hacer el Schema evolution.
- Podemos usar “spark.databricks.delta.schema.autoMerge= True” en Spark configuration
 - Usar con cuidado porque no te avisa de schema mismatches.
- Se usa comúnmente para añadir nuevas columnas o cambiar el tipo de datos (no-nullable a nullable, Bytetype → Shortype → IntegerType)

Ejercicios

Abrir 3 – Spark_Delta en Databricks

- Realiza hands on #2



Create TABLE

```
CREATE TABLE [ IF NOT EXISTS ] table_identifier
  [ ( col_name1 col_type1 [ COMMENT col_comment1 ], ... ) ]
  USING data_source
  [ OPTIONS ( key1=val1, key2=val2, ... ) ]
  [ PARTITIONED BY ( col_name1, col_name2, ... ) ]
  [ CLUSTERED BY ( col_name3, col_name4, ... )
    [ SORTED BY ( col_name [ ASC | DESC ], ... ) ]
    INTO num_buckets BUCKETS ]
  [ LOCATION path ]
  [ COMMENT table_comment ]
  [ TBLPROPERTIES ( key1=val1, key2=val2, ... ) ]
  [ AS select_statement ]
```

Alter TABLE

- Añadir columnas:

```
ALTER TABLE table_identifier ADD COLUMNS ( col_spec [ , ... ] )
```

- Añadir y borrar particiones:

```
ALTER TABLE table_identif ALTER TABLE table_name RENAME TO table_name  
partition_spec ... ] )
```

```
ALTER TABLE table_identifier DROP [ IF EXISTS ] partition_spec  
[PURGE]
```

- Renombrar:

```
ALTER TABLE table_name RENAME TO table_name
```

Alter TABLE

- Modificar columnas:

```
ALTER TABLE table_identifier (ALTER|CHANGE) [COLUMN] alterColumnAction
```

```
ALTER TABLE table_identifier (ALTER|CHANGE) [COLUMN] alterColumnAction
```

alterColumnAction:

```
: TYPE dataType
```

```
: [COMMENT col_comment]
```

```
: [FIRST|AFTER colA_name]
```

```
: (SET | DROP\| NOT NULL
```

- Reemplazar columnas:

```
ALTER TABLE table_name REPLACE COLUMNS (col_name1 col_type1 [COMMENT col_comment1], ...)
```

Constraints

Delta tables soporta el standard SQL verificando la calidad e integridad de los datos:

- NOT NULL
- CHECK

```
CREATE TABLE events(  
    id LONG NOT NULL,  
    date STRING NOT NULL,  
    location STRING,  
    description STRING  
) USING DELTA;
```

```
ALTER TABLE events CHANGE COLUMN date DROP NOT NULL;
```

```
CREATE TABLE events(  
    id LONG,  
    date STRING,  
    location STRING,  
    description STRING  
) USING DELTA;
```

```
ALTER TABLE events CHANGE COLUMN id SET NOT NULL;
```

Constraints

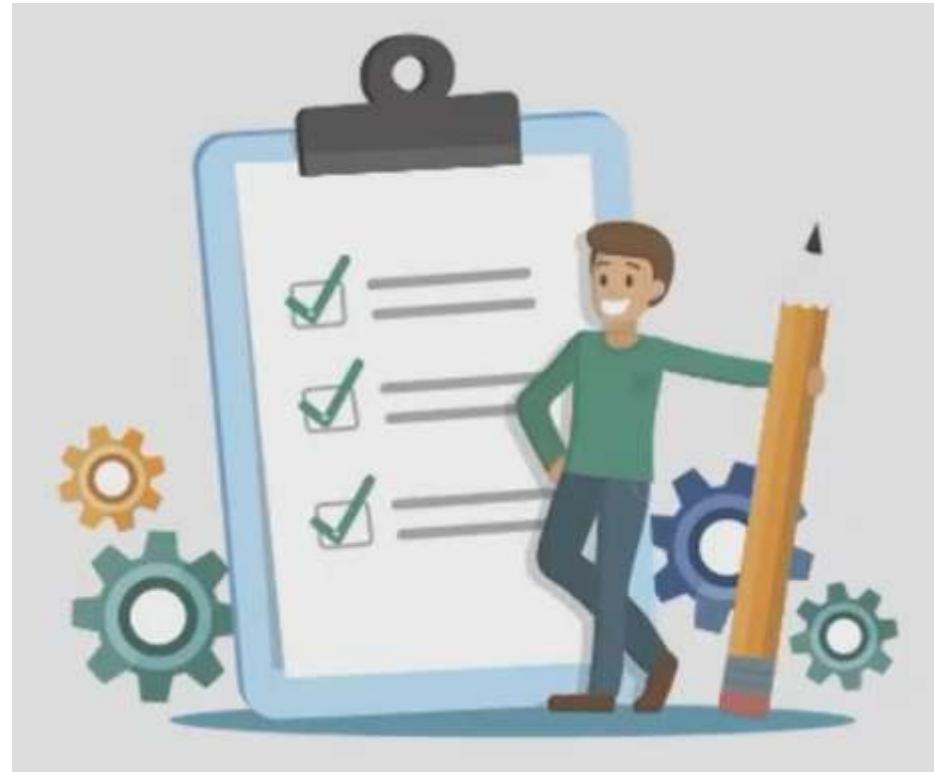
```
CREATE TABLE events(  
    id LONG NOT NULL,  
    date STRING,  
    location STRING,  
    description STRING  
) USING DELTA;  
  
ALTER TABLE events ADD CONSTRAINT dateWithinRange CHECK (date > '1900-01-01');  
ALTER TABLE events DROP CONSTRAINT dateWithinRange;
```

```
ALTER TABLE events ADD CONSTRAINT validIds CHECK (id > 1000 AND id < 999999);  
DESCRIBE DETAIL events;
```

Constraints

Abrir 3 – Spark_Delta en Databricks

Ejecutar el paso 4.1



Merge

Se puede realizar **upsert** data desde una tabla o dataframe de una tabla de Delta usando **merge operation**

```
MERGE INTO events
USING updates
ON events.eventId = updates.eventId
WHEN MATCHED THEN
    UPDATE SET events.data = updates.data
WHEN NOT MATCHED
    THEN INSERT (date, eventId, data) VALUES (date, eventId, data)
```

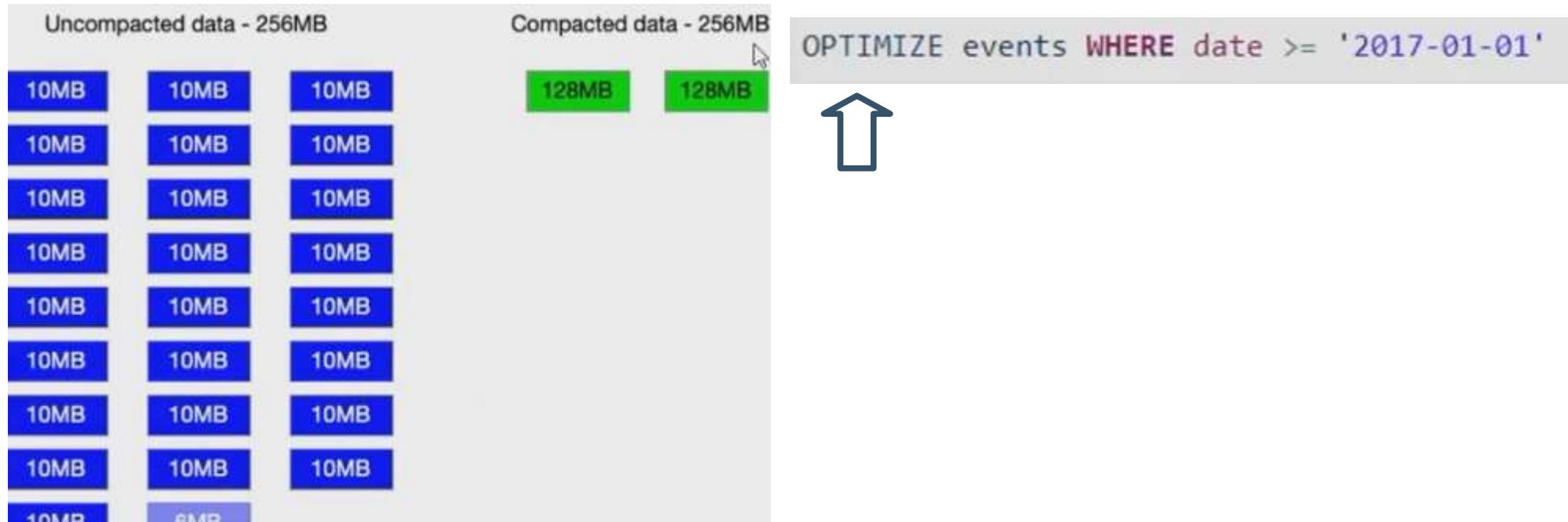
```
from delta.tables import *

deltaTable = DeltaTable.forPath(spark, "/data/events/")

deltaTable.alias("events").merge(
    updatesDF.alias("updates"),
    "events.eventId = updates.eventId") \
.whenMatchedUpdate(set = { "data" : "updates.data" } ) \
.whenNotMatchedInsert(values =
{
    "date": "updates.date",
    "eventId": "updates.eventId",
    "data": "updates.data"
})
.execute()
```

Optimización

Databricks puede mejorar la velocidad de lectura de las queries de una tabla haciendo coalescing de ficheros pequeños a ficheros grandes.



Data Skipping - Statistics

- Statistics information son colecciones automáticas de valores mínimos y máximos
- Cuando un nuevo dato es insertado dentro de Delta Table, file-level min/max statistics son para todas las tablas.
- Queries son mejoradas utilizando estos valores

```
SELECT * FROM table WHERE col <= 5
```

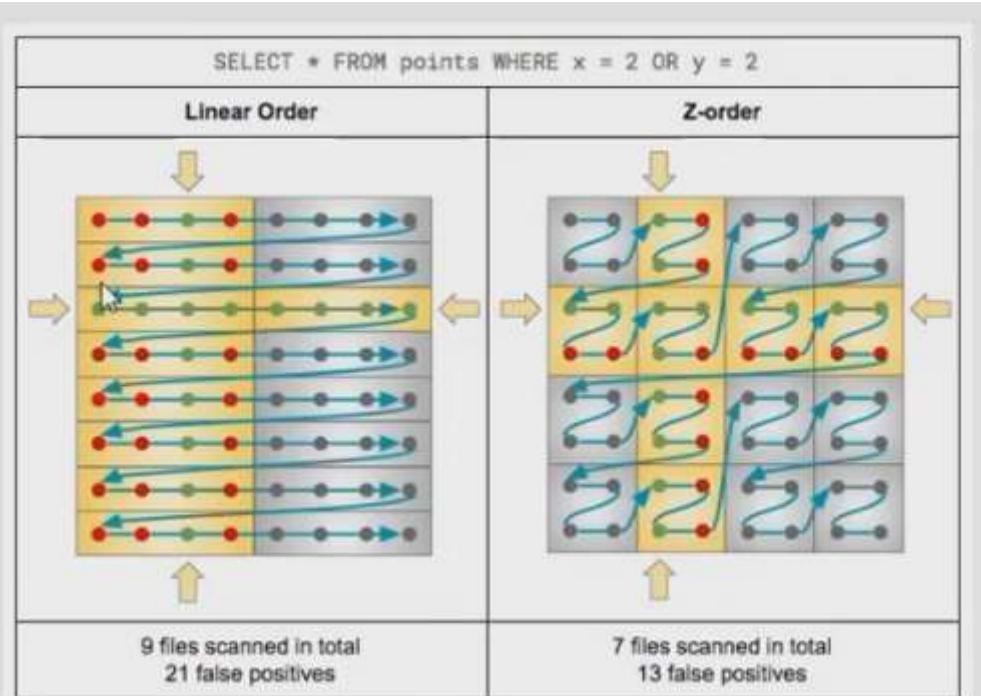


```
SELECT file_name FROM index  
WHERE col_min <= 5
```

file_name	col_min	col_max
data_file_1	6	8
data_file_2	3	10
data_file_3	1	4

Z - Ordering

- Técnica donde colocar la información relacionado en el mismo set de ficheros.



```
OPTIMIZE events
WHERE date >= current_timestamp() - INTERVAL 1 day
ZORDER BY (eventType)
```

Source: <https://databricks.com/blog/2018/07/31/processing-petabytes-of-data-in-seconds-with-databricks-delta.html>

