Apache Spark

Big Data Aplicado

¿Qué es Spark?

Apache Spark es un framework de computación distribuida, de código abierto (https://github.com/apache/spark) usado para procesamiento rápido de datos, realización de consultas y análisis de Big Data.

Spark vs Map Reduce (Hadoop)

- Trabaja 100 veces más rápido en computación in-memory.
- Tiene APIs para trabajar en Scala, Java, Python y R.
- Spark SQL.

https://spark.apache.org/

Características de Spark

- Computación en memoria principal
- Procesamiento distribuido usando paralelización
- Puede ser usado en variedad de clúster managers (Spark, Yarn, Mesos, etc)
- Tolerante a fallos
- Inmutable
- Lazy evaluation
- Cache & persistencia
- Inbuild-optimization usando DataFrames
- Soporta ANSI SQL

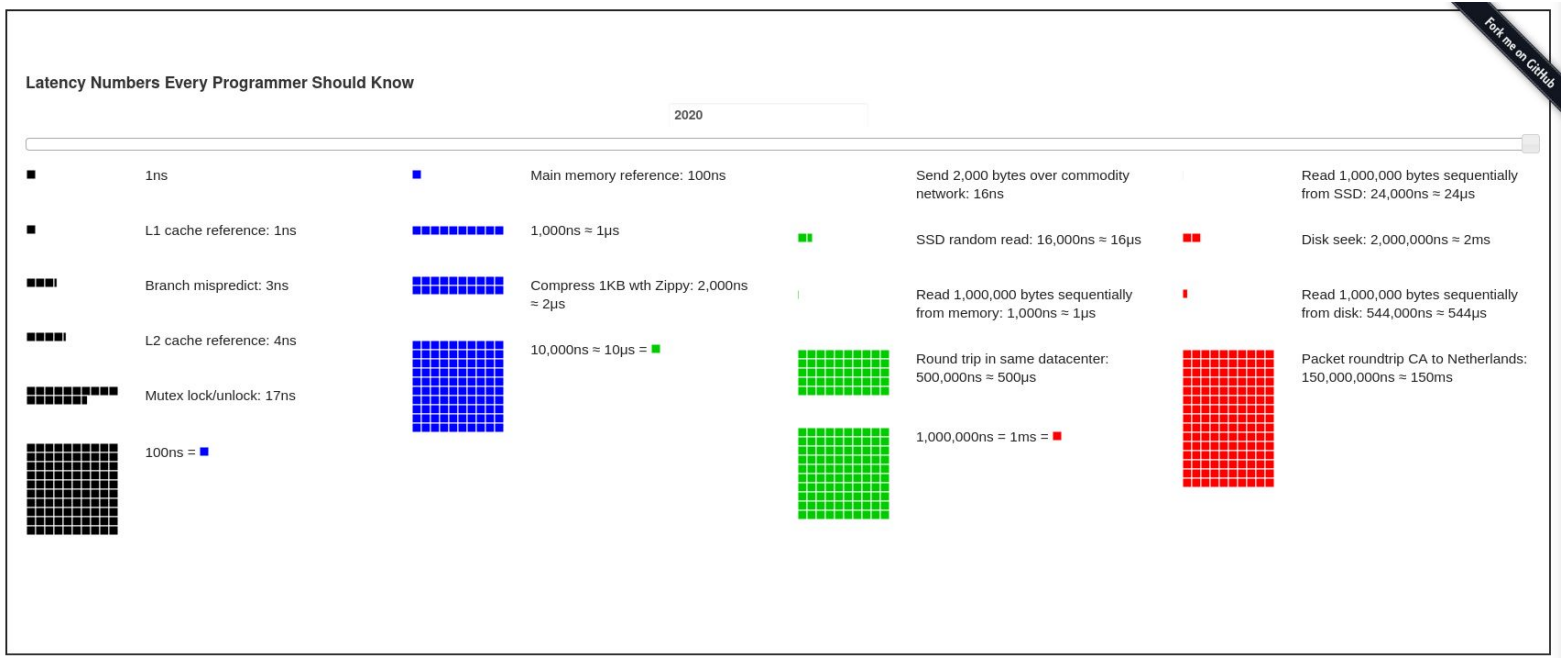
Computación distribuida

- **Fallos parciales (partial failure)**: errores en un subconjunto de máquinas en computación distribuida.
- **Latencia (latency)**: algunas operaciones tienen una latencia mucho mayor que otras en cuanto a la conexión en red.

Se debe reducir el número de conexiones en red, reduciendo el número de operaciones que necesitan dichas conexiones.

Comparación de latencias

https://colin-scott.github.io/personal_website/research/interactive_latency.html

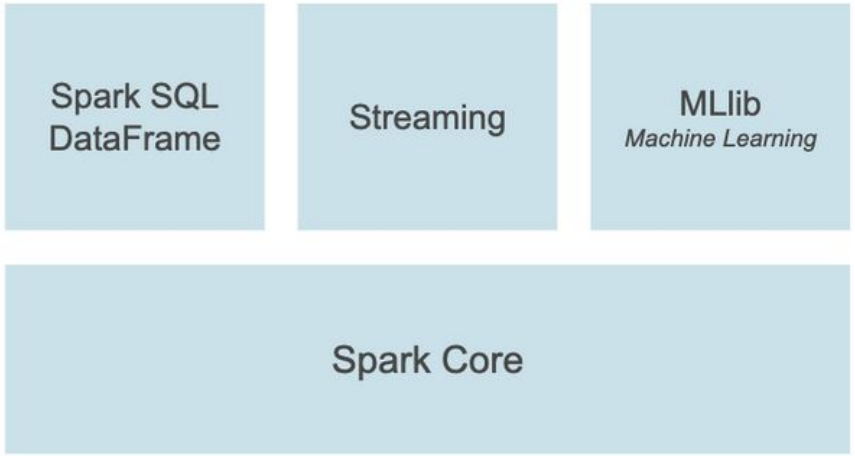


PySpark

PySpark es una interfaz para Apache Spark en Python.

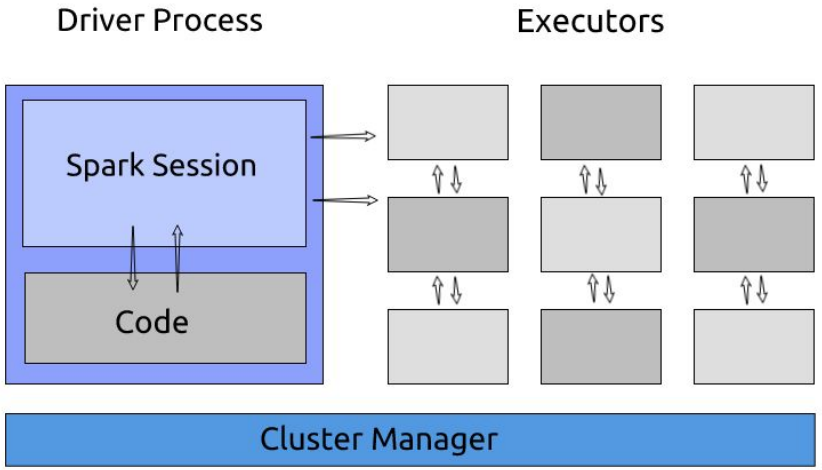
Nos permite:

- Escribir aplicaciones Spark usando la API de Python.
- Usar el PySpark Shell para analizar los datos de forma interactiva en un entorno distribuido.



Aplicaciones Spark

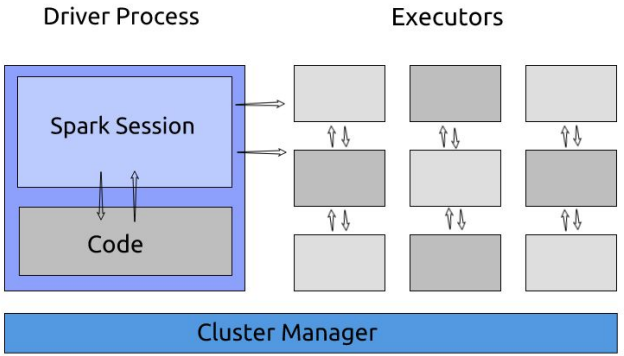
Una aplicación Spark es una instancia de **Spark Context** → La conjunción de un **proceso driver** + un conjunto de procesos **ejecutores**.



Aplicaciones Spark

- El Driver Process:
  - Mantiene información de la app Spark.
  - Organiza el trabajo en los distintos executors.
  - Enlazado con el código.

- Executor:
  - Ejecuta código asignado por el Driver.
  - Reporta el estado de su ejecución al Driver.



https://spark.apache.org/docs/latest/cluster-overview

Spark Session

Las aplicaciones PySpark comienzan inicializando la **SparkSession**, la cual es un punto de entrada de **PySpark**. Para realizar esta inicialización podemos ejecutar:

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.getOrCreate()
```

Spark Session

```
spark = SparkSession.builder\
    .master("local")\                    → URL de conexión
    .appName("Colab")\                   → Nombre de la aplicación (Spark UI)
    .config('spark.ui.port', '4050')\    → (key=None, value=None, conf=None)
    .getOrCreate()                       → Get de la sesión si existe, si no, crea una nueva
```

Spark Context

Representa la conexión a un cluster Spark y puede ser usado para crear RDD (Resilient Distributed Dataset) y otras variables en dicho clúster.

https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.SparkContext.html#pyspark.SparkContext

conf = SparkConf().setAppName(appName).setMaster(master)

sc = SparkContext(conf=conf)

Conceptos básicos de una aplicación Spark

Application → Un programa de usuario construido utilizando la API de Spark. Está compuesto de un programa “driver” y varios “executors”.

SparkSession → Un objeto que proporciona un punto de entrada para interactuar con la funcionalidad subyacente de Spark y permite programar en Spark con sus API. En un shell interactivo de Spark, el controlador de Spark instancia automáticamente un SparkSession para ti, mientras que en una aplicación de Spark, debes crear un objeto SparkSession tú mismo.

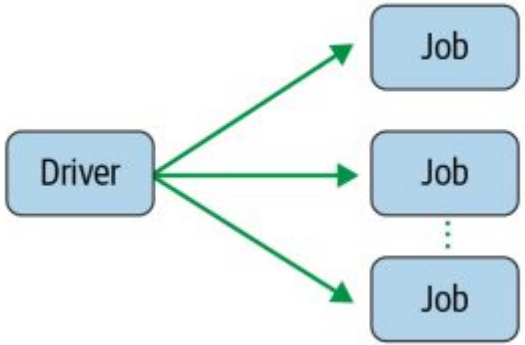
Job → Una computación paralela que consta de múltiples tareas que se inicia en respuesta a una acción de Spark (por ejemplo, save(), collect()).

Stage → Cada trabajo se divide en conjuntos más pequeños de tareas llamadas etapas que dependen entre sí.

Task → Una unidad única de trabajo o ejecución que se enviará a un ejecutor de Spark.

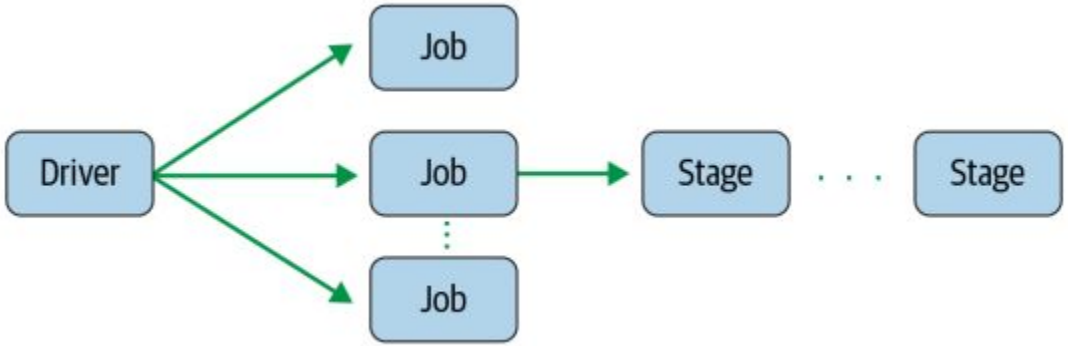
Spark Jobs

Durante sesiones interactivas con las shells de Spark, el controlador convierte tu aplicación de Spark en uno o más trabajos de Spark . Luego transforma cada trabajo en un DAG (grafo acíclico dirigido). Esto, en esencia, es el plan de ejecución de Spark, donde cada nodo dentro de un DAG puede ser una o varias etapas de Spark.



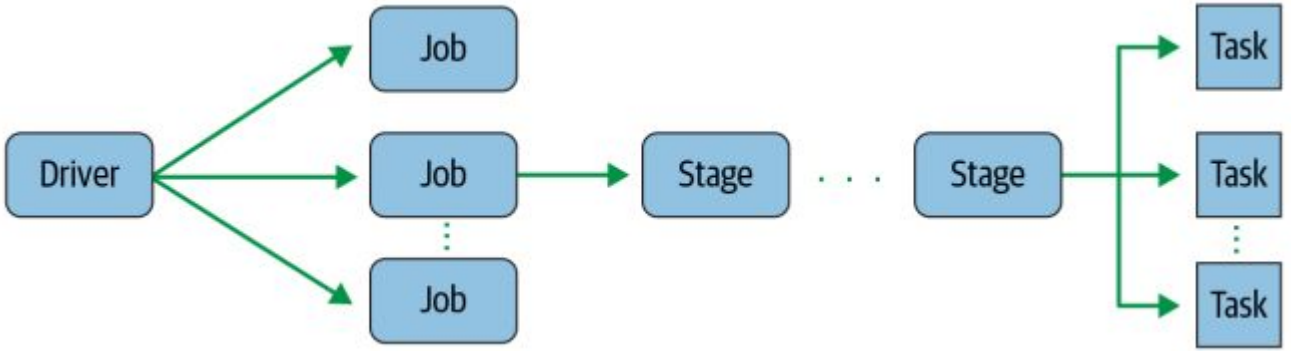
Spark Stages

Como parte de los nodos del DAG, se crean etapas en función de las operaciones que pueden realizarse en serie o en paralelo. No todas las operaciones de Spark pueden ocurrir en una sola etapa, por lo que pueden dividirse en múltiples etapas.



Spark Tasks

Cada etapa está compuesta por tareas de Spark (una unidad de ejecución), las cuales se distribuyen entre cada ejecutor de Spark; cada tarea se asigna a un núcleo único y trabaja en una única partición de datos. Por lo tanto, un ejecutor con 16 núcleos puede tener 16 o más tareas trabajando en 16 o más particiones en paralelo, lo que hace que la ejecución de las tareas de Spark sea excepcionalmente paralela.



Transformations vs Actions

Transformation → Devuelve un nuevo RDD como resultado de su aplicación.

Action → Calcula un resultado basado en un RDD, y lo devuelve o lo almacena en un sistema de almacenamiento (Ej: HDFS).

Transformation → **Lazy** (no se computa hasta que no se necesita)

Action → **Eager** (se computa inmediatamente)

La forma de **limitar la comunicación en red** de nuestro cluster es mediante este uso de transformaciones y acciones, sabiendo que se ejecutan de forma lazy/eager.

Transformations vs Actions

Tenemos el siguiente código python, haciendo uso de Spark:

```
strings = spark.read.text("../EJEMPLO.txt")

filtered = strings.filter(strings.value.contains("Hola"))

filtered.count()
```

En el código anterior, hasta que no se ejecuta el .count() de la tercera línea, no se obtiene el DataFrame original.

Transformations	Actions
orderBy()	show()
groupBy()	take()
filter()	count()
select()	collect()
join()	save()

Transformations

Una de las mayores ventajas de que las operaciones de transformación en RDD sean Lazy es la siguiente:

```
val lastYearsLogs: RDD[String] = ...

val firstLogsWithErrors = lastYearsLogs.filter(_.contains("ERROR")).take(10)
```

En este caso, la ejecución del filter es lazy, por lo que no se calculará el RDD hasta terminar el take de 10 elementos. Esto hace que nuestro proceso de coger los primeros 10 elementos haga que sólo se filtren esos 10 elementos en el filter, haciendo mucho más eficiente el proceso.

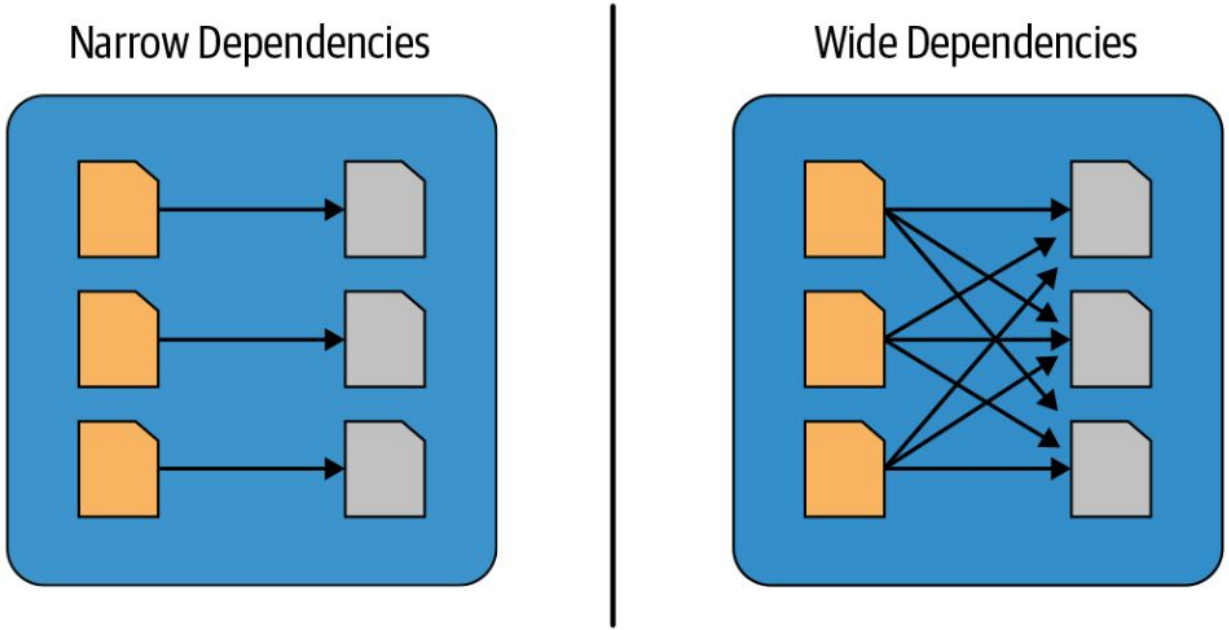
Transformaciones Narrow y Wide

Las transformaciones son operaciones que Spark **evalúa de manera perezosa**. Una gran ventaja del esquema de evaluación perezosa es que Spark puede inspeccionar tu consulta computacional y determinar cómo puede optimizarla. Esta optimización puede hacerse ya sea uniendo o canalizando algunas operaciones y asignándolas a una etapa, o dividiéndolas en etapas al determinar qué operaciones requieren una redistribución o intercambio de datos entre clústeres.

Las transformaciones pueden clasificarse como aquellas que tienen dependencias estrechas o dependencias amplias. Cualquier transformación donde una sola partición de salida pueda calcularse a partir de una sola partición de entrada es una transformación estrecha. Por ejemplo, en el fragmento de código anterior, filter() y contains() representan transformaciones estrechas porque pueden operar en una sola partición y producir la partición de salida resultante sin ningún intercambio de datos.

Sin embargo, groupBy() u orderBy() instruyen a Spark a realizar transformaciones amplias (wide), donde se lee, combina y escribe en disco datos de otras particiones. Dado que cada partición tendrá su propio recuento de la palabra que contiene la palabra "Hola" en su fila de datos, un recuento (groupBy()) forzará una redistribución de datos desde cada una de las particiones del ejecutor a lo largo del clúster. En esta transformación, orderBy() requiere la salida de otras particiones para calcular la agregación final.

Transformaciones Narrow y Wide



Hadoop vs Spark

	Hadoop	Spark
Almacenamiento	HDFS	-
MapReduce	Sí	Sí: Implementación propia
Velocidad	Rápido	10-100 veces más rápido
Resource management	YARN	Standalone, YARN, Mesos

RDD (Resilient Distributed Dataset)

Es la principal abstracción de Spark → Colección de elementos particionados a través de nodos dentro de un clúster, que pueden ser operados en paralelo.

Un RDD puede ser creado:

- Cargando un fichero del sistema de ficheros de Hadoop
- Desde una collection creada en Scala/Python

Creación de un RDD a partir de una lista en python:

```
data = [1, 2, 3, 4, 5]

distData = sc.parallelize(data)
```

RDD (Resilient Distributed Dataset)

Hay 3 características asociadas a los RDD:

- Dependencias
- Particiones
- Función de cómputo (Partition => Iterator[T])

RDD (Resilient Distributed Dataset)

Dependencias:

Lista de dependencias que indica a Spark cómo se construye un RDD con sus entradas de datos.

Cuando es necesario reproducir resultados, Spark puede recrear un RDD a partir de estas dependencias y replicar operaciones en él. Esta característica proporciona a los RDD resiliencia.

RDD (Resilient Distributed Dataset)

Particiones:

Las particiones proporcionan a Spark la capacidad de dividir el trabajo para paralelizar la computación en particiones a lo largo de los ejecutores. En algunos casos, por ejemplo, al leer desde HDFS, Spark utilizará información de localidad para enviar el trabajo a ejecutores cercanos a los datos. De esta manera, se transmite menos datos a través de la red.

RDD (Resilient Distributed Dataset)

Función de cómputo:

Un RDD tiene una función de cálculo que produce un Iterator[T] para los datos que se almacenarán en el RDD.

RDD

```
// Creación de un RDD a partir de una lista de tuplas
dataRDD = sc.parallelize([("Brooke", 20), ("Denny", 31), ("Jules", 30), ("TD", 35), ("Brooke", 25)])
// Cálculo de la media de las edades de las tuplas
agesRDD = (dataRDD
.map(lambda x: (x[0], (x[1], 1)))
.reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1]))
.map(lambda x: (x[0], x[1][0]/x[1][1]))).collect()

print(agesRDD)
```

DataFrames

Inspirados en la estructura, el formato y algunas operaciones específicas de los DataFrames de pandas, los DataFrames de Spark son como tablas distribuidas en memoria con columnas nombradas y esquemas, donde cada columna tiene un tipo de dato específico: entero, cadena, array, mapa, real, fecha, marca de tiempo, etc. A la vista humana, un DataFrame de Spark es como una tabla.

Id (Int)	First (String)	Last (String)	Url (String)	Published (Date)	Hits (Int)	Campaigns (List[Strings])
1	Jules	Damji	https://tinyurl.1	1/4/2016	4535	[twitter, LinkedIn]
2	Brooke	Wenig	https://tinyurl.2	5/5/2018	8908	[twitter, LinkedIn]
3	Denny	Lee	https://tinyurl.3	6/7/2019	7659	[web, twitter, FB, LinkedIn]
4	Tathagata	Das	https://tinyurl.4	5/12/2018	10568	[twitter, FB]

DataFrames: Data types

Data type	Value assigned in Python	API to instantiate
ByteType	int	DataTypes.ByteType
ShortType	int	DataTypes.ShortType
IntegerType	int	DataTypes.IntegerType
LongType	int	DataTypes.LongType
FloatType	float	DataTypes.FloatType
DoubleType	float	DataTypes.DoubleType
StringType	str	DataTypes.StringType
BooleanType	bool	DataTypes.BooleanType
DecimalType	decimal.Decimal	DecimalType

DataFrames: Creación de un DataFrame

data_df = spark.createDataFrame([("Brooke", 20), ("Denny", 31), ("Jules", 30), ("TD", 35), ("Brooke", 25)], ["name", "age"])

DataFrames: Definición del schema

1) **Podemos definir un schema programando la estructura:**

```
from pyspark.sql.types import *
schema = StructType([StructField("author", StringType(), False),
StructField("title", StringType(), False),
StructField("pages", IntegerType(), False)])
```

DataFrames: Definición del schema

2) **Podemos definir un schema usando Data Definition Language (DDL):**

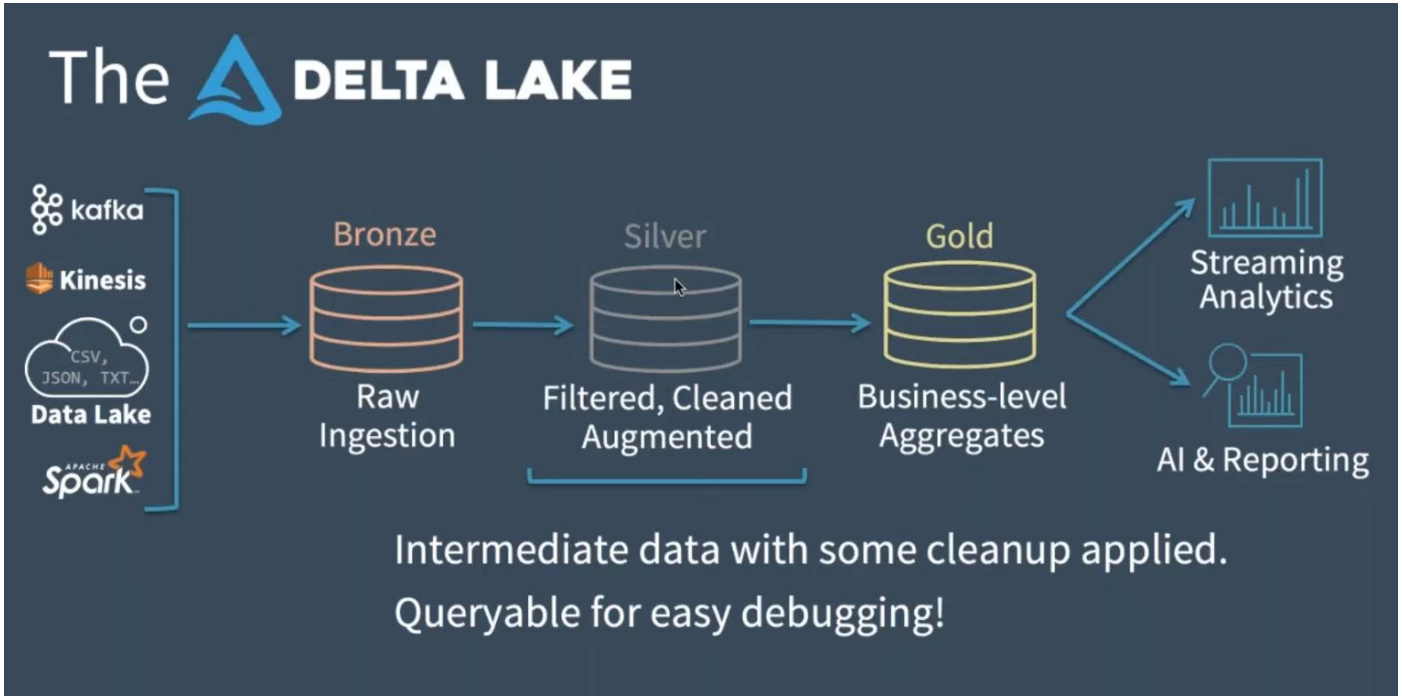
```
from pyspark.sql.types import *

schema = "author STRING, title STRING, pages INT"
```

DataFrames: Data types

Data type	Value assigned in Python	API to instantiate
`BinaryType`	`bytearray`	`BinaryType()`
`TimestampType`	`datetime.datetime`	`TimestampType()`
`DateType`	`datetime.date`	`DateType()`
`ArrayType`	List, tuple, or array	`ArrayType(dataType, [nullable])`
`MapType`	`dict`	`MapType(keyType, valueType, [nullable])`
`StructType`	List or tuple	`StructType([fields])`
`StructField`	A value type corresponding to the type of this field	`StructField(name, dataType, [nullable])`

Ejemplo de Delta Lake



Formatos de datos

JSON y **CSV** son muy comunes en almacenamiento de datos pero no fueron diseñados para almacenamiento masivo.

JSON → Es demasiado CPU-intensive (debido a la anidación de datos)

Ambas opciones utilizan formato de texto, por lo que son muy legibles para el ser humano, pero no son tan eficientes como las opciones binarias.

Avro, **ORC** y **Parquet** son algunos de estos ejemplos.

Row vs Columnar

Hay dos maneras en las que podemos organizar los datos: por filas y por columnas.

1. **Row** → Los datos están organizados en **registros.**

Escrituras rápidas vs **Lecturas** lentas.

Row vs Columnar

Hay dos maneras en las que podemos organizar los datos: por filas y por columnas.

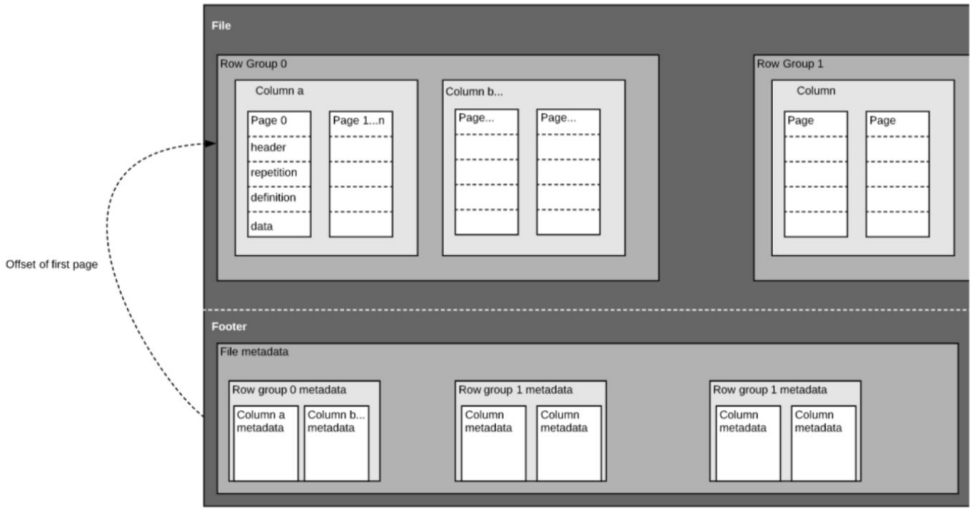
2. **Columnar** → Los valores de cada columna están almacenados juntos. Los ítems están agrupados y almacenados de forma conjunta.

Escrituras lentas vs **Lecturas** rápidas.

Parquet https://parquet.apache.org/

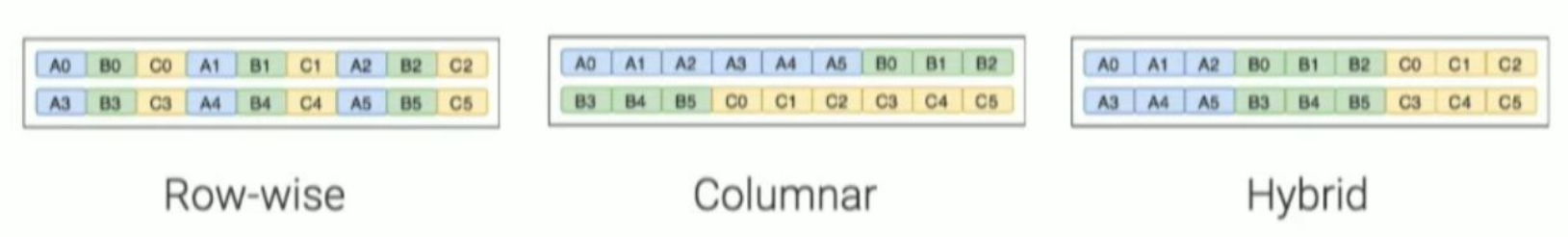
Diseñado para acceso y almacenamiento eficiente, además de facilitar una compresión eficiente y schemas.

Compatible con múltiples lenguajes de programación: Scala, Python, Java, C++...



Parquet

Cómo se almacenan los datos en disco (físicamente)



Parquet

Row-wise

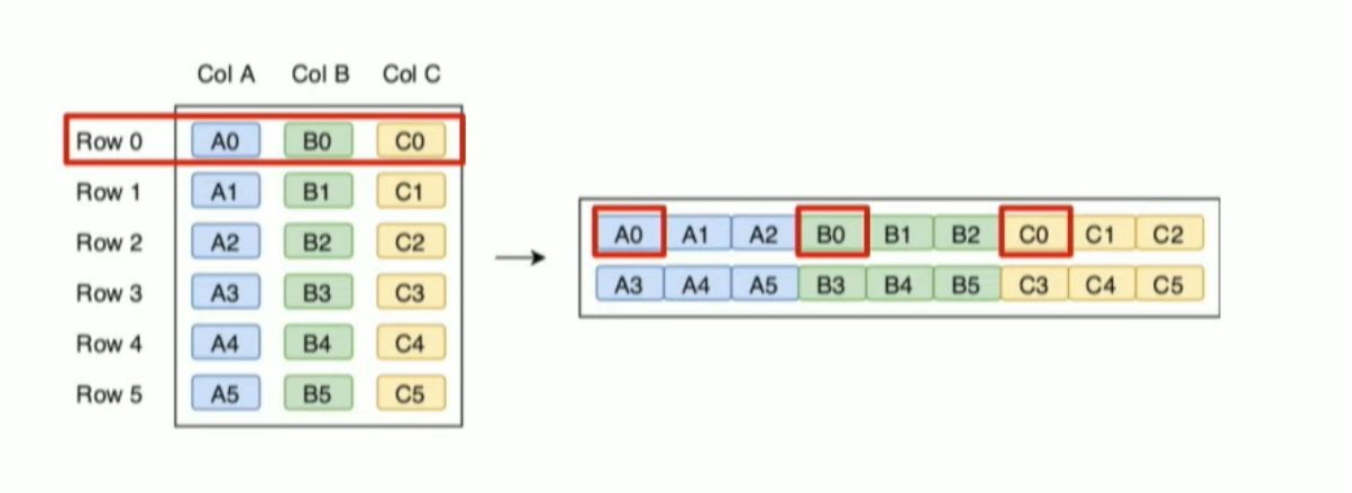
- Utiliza particionado horizontal
- OLTP → INSERTS, DELETES
- No OLAP → SELECTS de columnas específicas

Columnar

- Particionado vertical
- No OLTP
- OLAP
- Posibilidad de compresión → Mismo tipo de valor seguido

Parquet

Hybrid



Parquet

Características de Parquet

- Formato de archivo gratuito y de código abierto.
- Independiente del lenguaje de programación.
- Formato basado en columnas - los archivos se organizan por columna, en lugar de por fila, lo que ahorra espacio de almacenamiento y acelera las consultas analíticas.
- Se utiliza para casos de uso de análisis (OLAP), típicamente en combinación con los sistemas de base de datos OLTP tradicionales.
- Alta eficiencia en la compresión y descompresión de datos.
- Admite tipos de datos complejos y estructuras de datos anidados avanzadas.

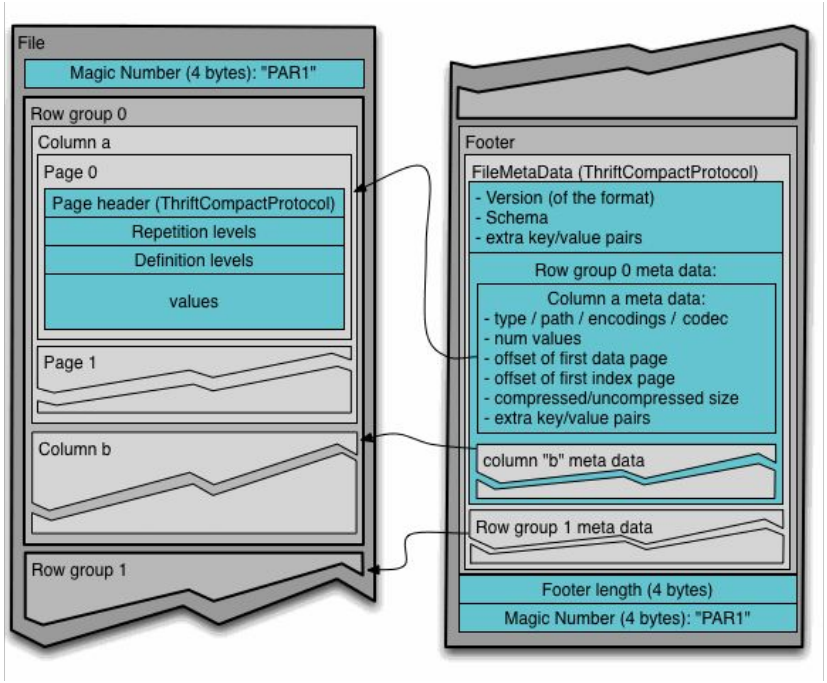
Parquet

Ventajas de Parquet

- Ideal para almacenar grandes volúmenes de datos de cualquier tipo (tablas de datos estructurados, imágenes, videos, documentos).
- Ahorra espacio en el almacenamiento en la nube al utilizar una compresión columna-única altamente eficiente y esquemas de codificación flexibles para columnas con diferentes tipos de datos.
- Aumenta el rendimiento y el throughput de datos mediante técnicas como el salto de datos, donde las consultas que buscan valores específicos de columna no necesitan leer toda la fila de datos.
- Apache Parquet se implementa utilizando el algoritmo de particionamiento y ensamblaje de registros, lo que permite adaptarse a las complejas estructuras de datos que se pueden utilizar para almacenar los datos. Parquet está optimizado para trabajar con datos complejos en grandes cantidades y ofrece diferentes formas de compresión y codificación eficientes de datos. Esta aproximación es especialmente útil para consultas que necesitan leer columnas específicas de una tabla grande. Parquet solo puede leer las columnas necesarias, lo que minimiza en gran medida el IO.

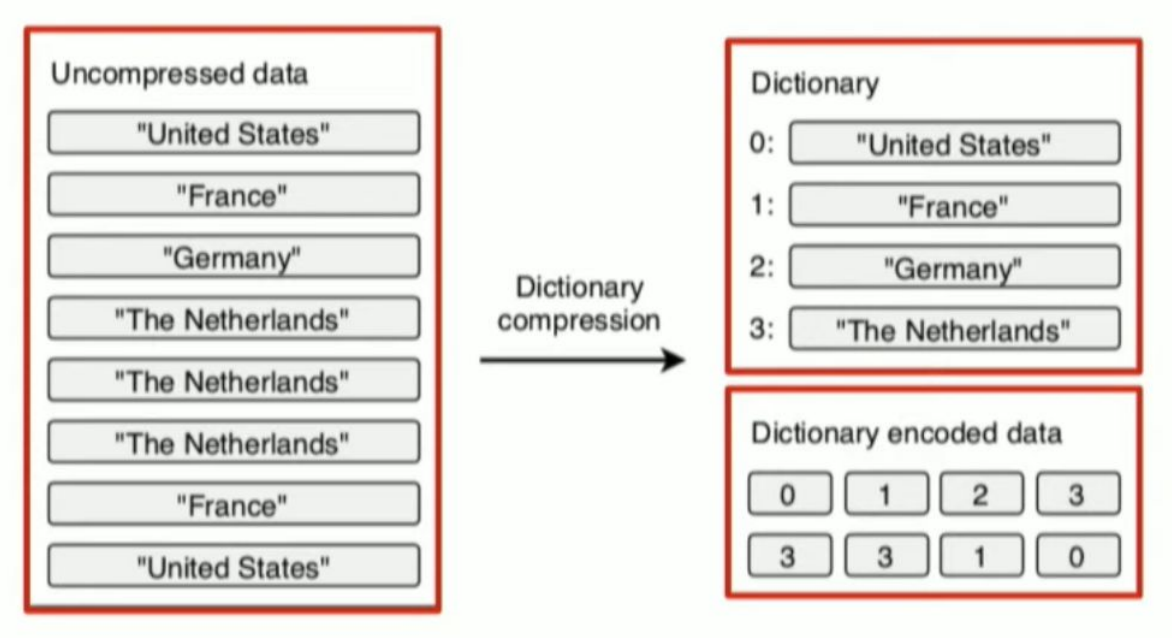
Parquet: organización de los datos

- Row-groups (128 MB)
- Column chunks
- Pages (1 MB por defecto)
  - Metadatos:
    - Min
    - Max
    - Count



Parquet: Optimizaciones

- **Dictionary encoding**



Parquet: Optimizaciones

- **Particionado**

df.write.partitionBy(‘date’).parquet(...)

./data/date=2024-03-01/...

./data/date=2024-03-02/...

./data/date=2024-03-03/part-00000-...-123h12e2345d.c000.snappy.parquet

Parquet: Optimizaciones

- **Compactación de muchos ficheros pequeños**

Se puede realizar de forma manual:

```
df.repartition(numPartitions).write.parquet(...)

df.coalesce(numPartitions).write.parquet(...)
```

Parquet

Resumen

- Data compression
- Columnar storage
- Language agnostic
- Open-source
- Complex data types

AVRO https://avro.apache.org/docs/1.11.1/

- Estructuras de datos ricas (uso de schemas).
- Formato de datos binario compacto y rápido.
- Ficheros contenedores que almacenan datos persistentes.
- Remote procedure call (RPC).
- Integración con lenguajes de programación dinámicos.
- Ficheros divisibles
- Soporta schemas evolutivos
- Varias opciones de compresión de ficheros (bzip2, uncompressed, snappy)

¿Cuándo usar AVRO?

- Cuando se tiene una gran cantidad de operaciones de escritura.
- Cuando la velocidad de escritura y la evolución de los schemas son críticos.

AVRO

Lectura de AVRO en DataFrames:

```
df = (spark.read.format("avro").load("/path/to/files/*"))

df.show(truncate=False)
```

AVRO

Creación de vista en SQL con formato AVRO:

CREATE OR REPLACE TEMPORARY VIEW test_tbl

USING avro

OPTIONS (

path "*/path/to/files/")**

AVRO

De DataFrame a ficheros AVRO:

```
(df.write
.format("avro")
.mode("overwrite")
.save("/tmp/data/avro/df_avro"))
```

Esto genera una carpeta en tmp con un conjunto de ficheros comprimidos AVRO.

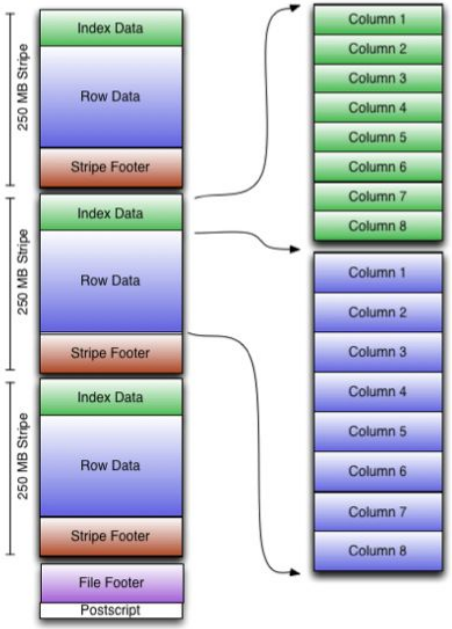
ORC (Optimized Row Columnar)

- **Index Data**: Ubicación de datos además de secciones de índice que se utilizan para acelerar la búsqueda de datos en el archivo.

- **Row Data**: Donde se almacenan los datos. Formato columnar (valores de la misma columna se almacenan juntos).

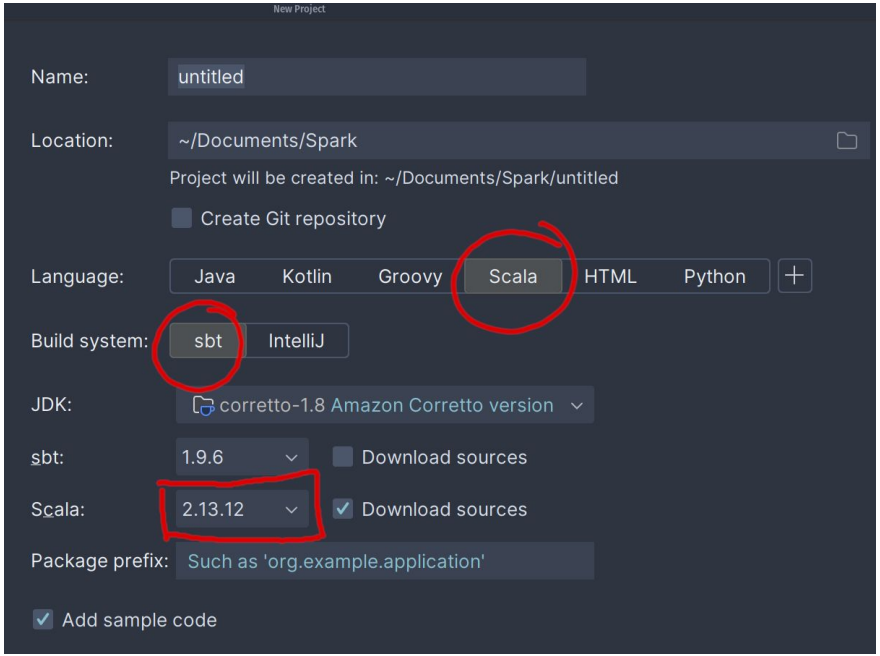
- **Stripe Footer**: Contiene información acerca del stripe y su ubicación en el archivo → Lecturas eficientes de los stripes.

https://cwiki.apache.org/confluence/display/hive/languagemanual+orc



Creación y configuración de proyecto Spark

Creación de proyecto en Intellij:



Elegimos Scala como lenguaje de programación (también podríamos usar Python y crear un proyecto en PySpark).

Para gestionar los paquetes de nuestro proyecto tenemos dos opciones: usar Intellij y gestionar de forma manual los paquetes o usar sbt (https://www.scala-sbt.org/)

Por último, elegimos la versión de Scala < 3, para que sea compatible con nuestra versión de Spark.

Creación y configuración de proyecto Spark

Configuración de build.sbt:

```
ThisBuild / scalaVersion := "2.13.12"

val sparkVersion = "3.3.2"

libraryDependencies ++= Seq(
  "org.apache.spark" %% "spark-core" % sparkVersion,
  "org.apache.spark" %% "spark-sql" % sparkVersion,
  "org.apache.spark" %% "spark-mllib" % sparkVersion,
)
```

A la primera línea de la imagen (que ya está configurada por defecto al crear el proyecto en el paso anterior), añadimos una **variable** con el valor de la versión de Spark que vamos a utilizar.

Añadimos también a las dependencias los paquetes de spark que vamos a utilizar en el proyecto (en nuestro caso sólo serían necesarios los dos primeros, **core** y **sql**)