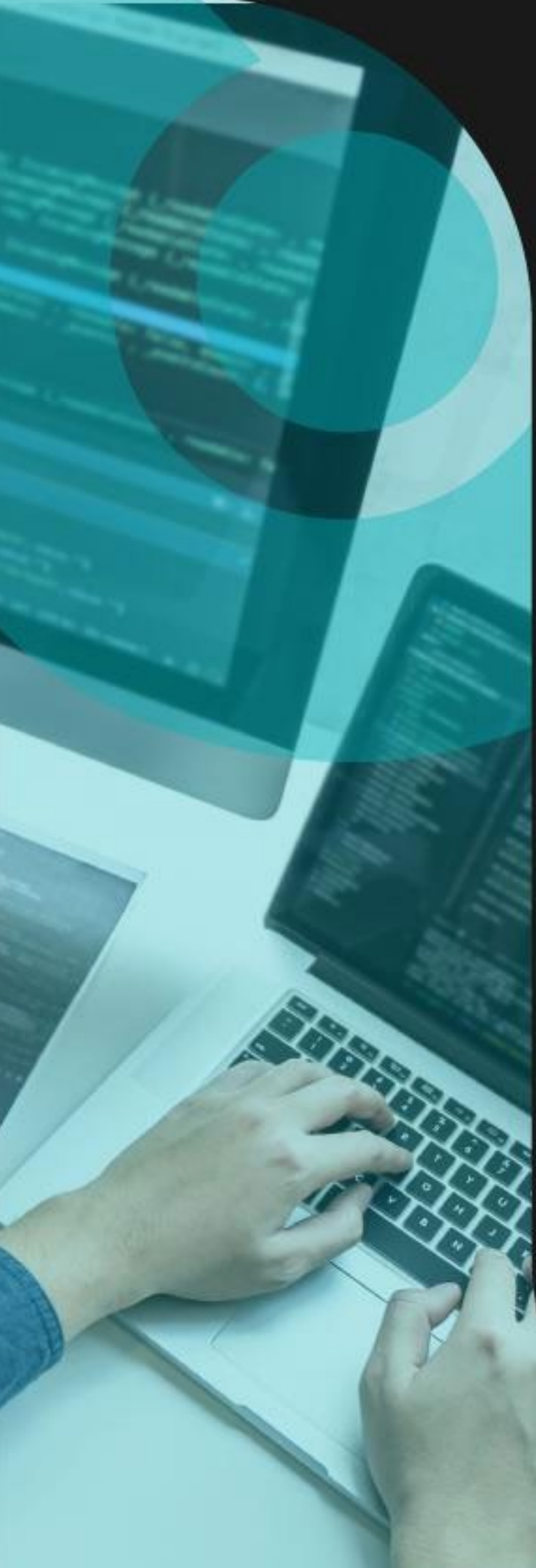


AWAKELAB

# BASECAMP

## Ciencia de Datos

**Módulo: Fundamentos del Big Data**

---

**Aprendizaje Esperado**

---

3. Manipular grandes volúmenes de datos utilizando Spark SQL para resolver un problema

---

## Spark SQL

Spark presenta un módulo de programación para el procesamiento de datos estructurados llamado Spark SQL. Proporciona una abstracción de programación llamada DataFrame y puede actuar como un motor de consulta SQL distribuido.

¿Qué es SQL?

**SQL** es un acrónimo en inglés para **Structured Query Language**. Un **Lenguaje de Consulta Estructurado**. Un tipo de **lenguaje de programación** que te permite manipular y descargar datos de una base de datos. Tiene capacidad de hacer cálculos avanzados y álgebra. Es utilizado en la mayoría de empresas que almacenan datos en una base de datos. Ha sido y sigue siendo el lenguaje de programación más usado para bases de datos relacionales.

## Características de Spark SQL

Las siguientes son las características de Spark SQL:

- **Integrado** : mezcla a la perfección consultas SQL con programas Spark. Spark SQL le permite consultar datos estructurados como un conjunto de datos distribuidos (RDD) en Spark, con API integradas en Python, Scala y Java. Esta estrecha integración facilita la ejecución de consultas SQL junto con algoritmos analíticos complejos.

- **Acceso unificado a datos** : cargue y consulte datos de una variedad de fuentes. Schema-RDD proporciona una interfaz única para trabajar de manera eficiente con datos estructurados, incluidas las tablas de Apache Hive, los archivos de parquet y los archivos JSON.

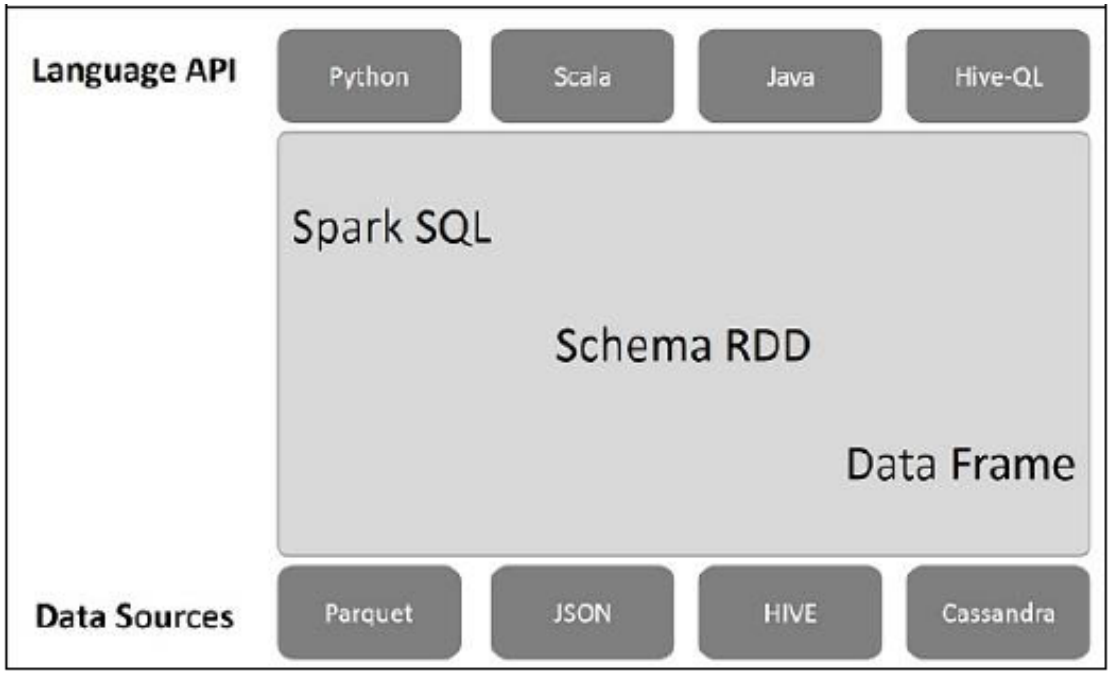
- **Compatibilidad con Hive** : ejecute consultas de Hive sin modificar en almacenes existentes. Spark SQL reutiliza la interfaz Hive y MetaStore, lo que le brinda compatibilidad total con los datos, consultas y UDF existentes de Hive. Simplemente instálelo junto con Hive.

- **Conectividad estándar** : conexión a través de JDBC u ODBC. Spark SQL incluye un modo de servidor con conectividad JDBC y ODBC estándar de la industria.

- **Escalabilidad** : use el mismo motor para consultas largas e interactivas. Spark SQL aprovecha el modelo RDD para admitir la tolerancia a fallas en la mitad de la consulta, lo que también le permite escalar a trabajos grandes. No se preocupe por usar un motor diferente para los datos históricos.

**Arquitectura Spark SQL**

La siguiente ilustración explica la arquitectura de Spark SQL:




Esta arquitectura contiene tres capas, a saber, Language API, Schema RDD y Data Sources.

- **API de idioma** : Spark es compatible con diferentes idiomas y Spark SQL. También es compatible con estos lenguajes: API (python, scala, java, HiveQL).

- **Schema RDD** − Spark Core está diseñado con una estructura de datos especial llamada RDD. Generalmente, Spark SQL funciona en esquemas, tablas y registros. Por lo tanto, podemos usar el Schema RDD como tabla temporal. Podemos llamar a este esquema RDD como marco de datos.

- **Fuentes de datos** : por lo general, la fuente de datos para Spark-Core es un archivo de texto, un archivo Avro, etc. Sin embargo, las fuentes de datos para Spark SQL son diferentes. Esos son el archivo Parquet, el documento JSON, las tablas HIVE y la base de datos Cassandra.

**¿Qué es SparkSession?**

SparkSession se introdujo en la versión Spark 2.0. Es un punto de entrada a la funcionalidad subyacente de Spark para crear mediante programación Spark RDD, DataFrame y DataSet. El objeto de SparkSession spark es la variable predeterminada disponible spark-shelly se puede crear mediante programación usando SparkSession un patrón de generador.

**Crear un esquema DataFrame**

SparkSession también proporciona varios métodos para crear un DataFrame y un DataSet de Spark. El siguiente ejemplo utiliza el createDataFrame() método que toma una lista de datos.

```
// Create DataFrame
val df = spark.createDataFrame(
    List(("Scala", 25000), ("Spark", 35000), ("PHP", 21000)))
df.show()


// Output
//+-----+-----+
//|   _1|   _2|
//+-----+-----+
//|Scala|25000|
//|Spark|35000|
//|  PHP|21000|
//+-----+-----+
```

**Funciones por el usuario - Trabajar con Spark SQL**

Usando SparkSession puede acceder a las capacidades de Spark SQL en Apache Spark. Para usar primero las funciones de SQL, debe crear una vista temporal en Spark. Una vez que tenga una vista temporal, puede ejecutar cualquier consulta ANSI SQL utilizando spark.sql()el método.

```
// Spark SQL

df.createOrReplaceTempView("sample_table")

val df2 = spark.sql("SELECT _1,_2 FROM sample_table")

df2.show()
```

Las vistas temporales de Spark SQL tienen un ámbito de sesión y no estarán disponibles si finaliza la sesión que las crea. Si desea tener una vista temporal que se comparta entre todas las sesiones y que se mantenga viva hasta que finalice la aplicación Spark, puede crear una vista temporal global usando createGlobalTempView()

**Interoperaciones RDD**

Los RDD admiten dos tipos de operaciones: *transformaciones* , que crean un nuevo conjunto de datos a partir de uno existente, y *acciones* , que devuelven un valor al programa controlador después de ejecutar un cálculo en el conjunto de datos. Por ejemplo, mapes una transformación que pasa cada elemento del conjunto de datos a través de una función y devuelve un nuevo RDD que representa los resultados. Por otro lado, reducees una acción que agrega todos los elementos del RDD mediante alguna función y devuelve el resultado final al programa controlador (aunque también existe un paralelo reduceByKeyque devuelve un conjunto de datos distribuido).

Todas las transformaciones en Spark son *perezosas*, ya que no calculan sus resultados de inmediato. En cambio, solo recuerdan las transformaciones aplicadas a algún conjunto de datos base (por ejemplo,

un archivo). Las transformaciones solo se calculan cuando una acción requiere que se devuelva un resultado al programa controlador. Este diseño permite que Spark funcione de manera más eficiente. Por ejemplo, podemos darnos cuenta de que un conjunto de datos creado a través mapde se usará en un reducey devolverá solo el resultado del reducecontrolador, en lugar del conjunto de datos mapeado más grande.

De forma predeterminada, cada RDD transformado se puede volver a calcular cada vez que ejecuta una acción en él. Sin embargo, también puede conservar un *RDD* en la memoria usando el método persist(o cache), en cuyo caso Spark mantendrá los elementos en el clúster para un acceso mucho más rápido la próxima vez que lo consulte. También hay soporte para RDD persistentes en el disco o replicados en múltiples nodos.

**Pasar funciones a Spark**

La API de Spark se basa en gran medida en pasar funciones en el programa del controlador para que se ejecuten en el clúster. Hay tres formas recomendadas de hacer esto:

- Expresiones lambda , para funciones simples que se pueden escribir como una expresión. (Lambdas no admite funciones de instrucciones múltiples o instrucciones que no devuelven un valor).
- Local defs dentro de la función que llama a Spark, para un código más largo.
- Funciones de nivel superior en un módulo.

**Comprender los cierres**

Una de las cosas más difíciles de Spark es comprender el alcance y el ciclo de vida de las variables y los métodos al ejecutar código en un clúster. Las operaciones RDD que modifican variables fuera de su alcance pueden ser una fuente frecuente de confusión. En el siguiente ejemplo, veremos el código que se usa foreach()para incrementar un contador, pero también pueden ocurrir problemas similares para otras operaciones.

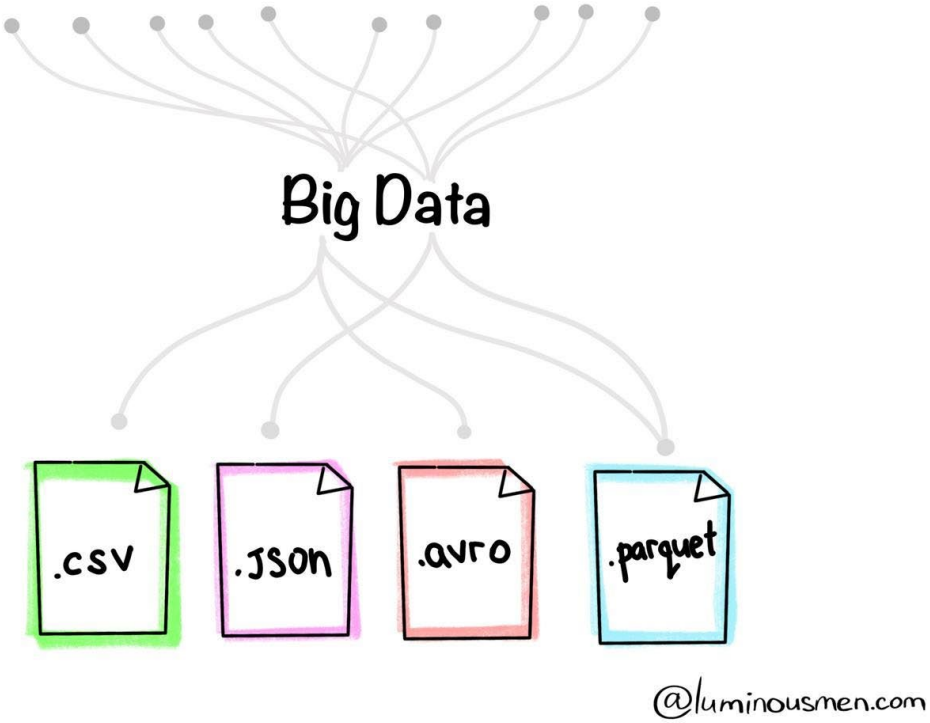
**Ejemplo**

Considere la suma del elemento RDD ingenuo a continuación, que puede comportarse de manera diferente dependiendo de si la ejecución se lleva a cabo dentro de la misma JVM. Un ejemplo común de esto es cuando se ejecuta Spark en localmodo ( --master = local[n]) en lugar de implementar una aplicación Spark en un clúster (por ejemplo, a través de Spark-submit to YARN):

```
pip install pyspark

from pyspark import SparkConf,SparkContext


conf = SparkConf().setAppName("app").setMaster('local')

sc = SparkContext(conf=conf)

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

distData = sc.parallelize(data)

distFile = sc.textFile("data.txt")
```

```
counter = 0

rdd = sc.parallelize(data)


# Wrong: Don't do this!!

def increment_counter(x):

    global counter

    counter += x
```

```
rdd.foreach(increment_counter)


print("Counter value: ", counter)
```




Apache Spark admite muchos formatos de datos diferentes, como el omnipresente formato CSV y el formato web amigable JSON. Los formatos más comunes utilizados principalmente para el análisis de big data son Apache Parquet y Apache Avro.

**JSON**

Los datos JSON (notación de objetos de JavaScript) se presentan como pares clave-valor en un formato parcialmente estructurado. JSON a menudo se compara con XML porque puede almacenar datos en un formato jerárquico. Ambos formatos son legibles por el usuario, pero los

documentos JSON suelen ser mucho más pequeños que XML. Por lo tanto, se usan más comúnmente en la comunicación de red, especialmente con el auge de los servicios web basados en REST.

Dado que muchos datos ya se transmiten en formato JSON, la mayoría de los lenguajes web inicialmente admiten JSON. Con este gran soporte, JSON se usa para representar estructuras de datos, formatos de intercambio para datos calientes y almacenes de datos fríos.

Muchos paquetes y módulos de transmisión admiten la serialización y deserialización de JSON. Si bien los datos contenidos en los documentos JSON se pueden almacenar en última instancia en formatos más optimizados para el rendimiento, como Parquet o Avro, sirven como datos sin procesar, lo cual es muy importante para el procesamiento de datos (si es necesario).

Pros y contras del formato :

- JSON admite estructuras jerárquicas, lo que simplifica el almacenamiento de datos relacionados en un solo documento y presenta relaciones complejas.
- La mayoría de los lenguajes proporcionan bibliotecas de serialización JSON simplificadas o soporte integrado para serialización/deserialización JSON.
- JSON admite listas de objetos, lo que ayuda a evitar la conversión caótica de listas a un modelo de datos relacional.
- JSON es un formato de archivo ampliamente utilizado para bases de datos NoSQL como MongoDB, Couchbase y Azure Cosmos DB.
- Soporte incorporado en la mayoría de las herramientas modernas;

- JSON consume más memoria debido a los nombres de columna repetibles.
- Poco soporte para caracteres especiales.
- JSON no es muy divisible.
- JSON carece de indexación.
- Es menos compacto en comparación con los formatos binarios.

**Parquet**

Dado que los datos se almacenan en columnas, pueden comprimirse mucho (los algoritmos de compresión funcionan mejor con datos con baja entropía de la información, que suele estar contenida en columnas) y pueden separarse. Los desarrolladores del formato afirman que este formato de almacenamiento es ideal para resolver problemas de Big Data.

A diferencia de CSV y JSON, los archivos de parquet son archivos **binarios** que contienen **metadatos** sobre su contenido. Por lo tanto, sin leer/analizar el contenido de los archivos, Spark simplemente puede confiar en los metadatos para determinar los nombres de las columnas, la compresión/codificación, los tipos de datos e incluso algunas características estadísticas básicas. Los metadatos de columna para un archivo de Parquet se almacenan al final del archivo, lo que permite una escritura rápida de un solo paso.

Parquet está optimizado para el paradigma *Write Once Read Many (WORM)* . Escribe lentamente pero lee increíblemente rápido, especialmente cuando solo accede a un subconjunto de columnas. El parquet es **una buena opción para cargas de trabajo pesadas al leer porciones de datos** . Para los casos en los que necesite trabajar con filas enteras de datos, debe usar un formato como CSV o AVRO.

Pros y contras del formato:

- El parquet es un *formato columnar* . Solo se recuperarán/leerán las columnas requeridas, esto *reduce la E/S del disco* . El concepto se llama *pushdown de proyección*.
- El esquema viaja con los datos, por lo que los datos se *describen a sí mismos*.
- Aunque está diseñado para HDFS, los datos se pueden almacenar en otros sistemas de archivos como GlusterFs o NFS.
- Solo archivos de parquet, lo que significa que es fácil trabajar, mover, hacer copias de seguridad y replicarlos.
- El soporte incorporado en Spark facilita tomar y guardar un archivo en el almacenamiento.

- El parquet proporciona una muy buena compresión de hasta un 75 % cuando se utilizan incluso formatos de compresión como Snappy.
- Como muestra la práctica, este formato es el más rápido para procesos de lectura intensiva en comparación con otros formatos de archivo.
- Parquet es muy adecuado para soluciones de almacenamiento de datos donde se requiere la agregación en una columna en particular sobre un gran conjunto de datos.
- Parquet se puede leer y escribir utilizando Avro API y Avro Schema (lo que da la idea de almacenar todos los datos sin procesar en formato Avro, pero todos los datos procesados en parquet).
- También proporciona *pushdown de predicados* , lo que reduce el costo adicional de transferir datos desde el almacenamiento al motor de procesamiento para el filtrado.

- El diseño basado en columnas te hace pensar en el esquema y los tipos de datos.
- Parquet no siempre tiene soporte integrado en otras herramientas además de Spark.
- No admite modificación de datos (los archivos Parquet son inmutables) y evolución de esquemas. Por supuesto, Spark sabe cómo combinar el esquema si lo cambia con el tiempo (debe especificar una opción especial mientras lee), pero solo puede cambiar algo en un archivo existente sobrescribiéndolo.

**Referencias**

[1] ¿Qué es SQL?

https://youtu.be/TCam1GMMjTg

[2] SQL spark

https://cloud.ibm.com/docs/AnalyticsEngine?topic=AnalyticsEngine-working-with-sql&locale=es#:~:text=La%20CLI%20de%20SQL%20Spark%20es%20una%20herramienta%20pr%C3%A1ctica%20para,de%20la%20l%C3%ADnea%20de%20mandatos.

[3] Official web site – Apache Spark

https://spark.apache.org/sql/

**Complementario**

[1] Qué es Apache spark

https://www.youtube.com/watch?v=WR9HnAdYOfs