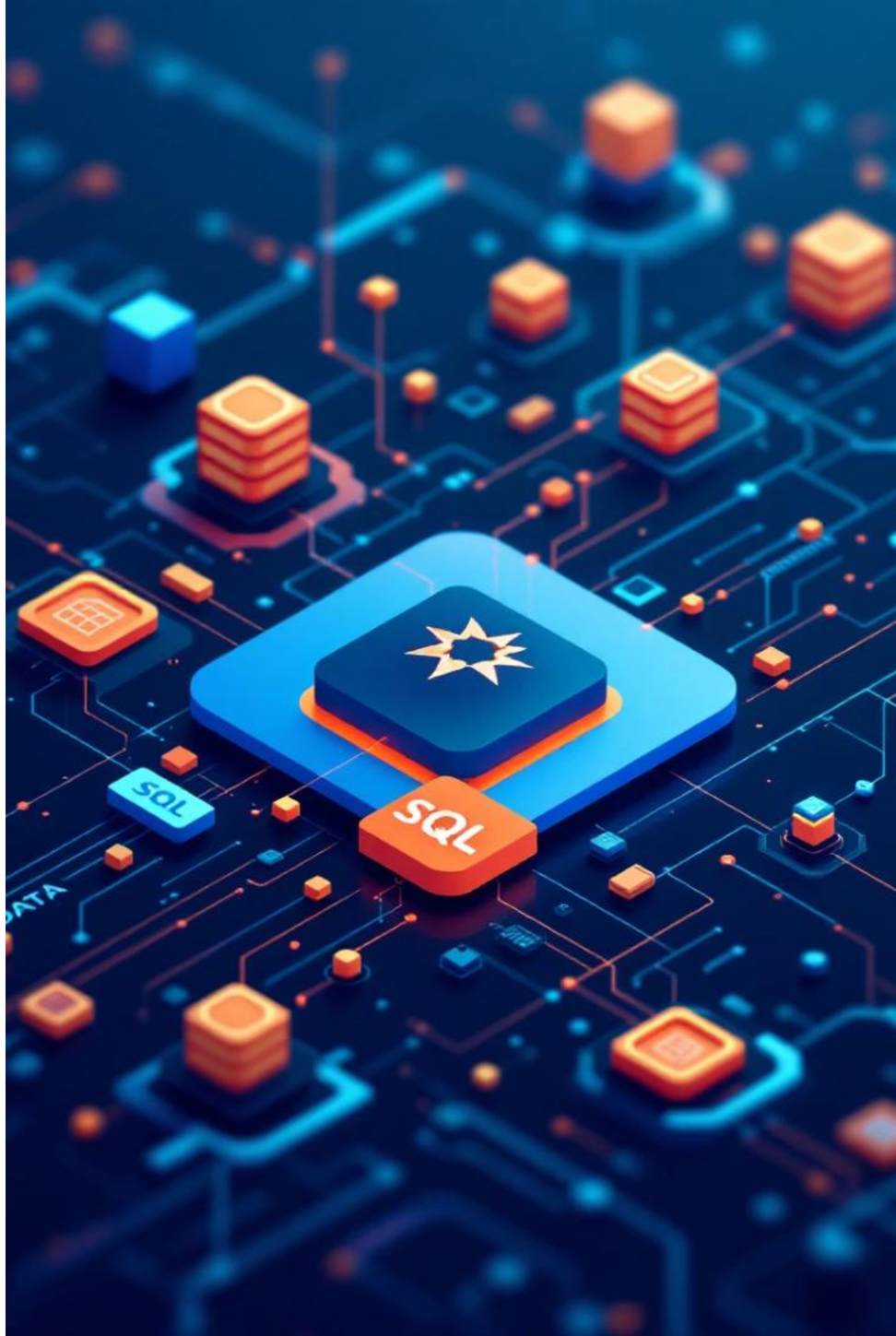
Procesamiento de Datos Estructurados con Spark SQL

Spark SQL es un módulo de Apache Spark diseñado para trabajar con datos estructurados y semiestructurados. Proporciona una API para ejecutar consultas SQL sobre datos almacenados en diversos formatos y fuentes, como Parquet, ORC, JSON, bases de datos JDBC, y más.

Este potente módulo permite a los usuarios combinar SQL con código en Python, Scala, Java y R, integrando el procesamiento distribuido de Spark con la flexibilidad del lenguaje SQL, lo que facilita enormemente el análisis de grandes volúmenes de datos estructurados.

por Kibernum Capacitación S.A.



Preguntas de Activación de Contenidos

1) ¿Qué es un RDD en Apache Spark y cómo se crea?

2) ¿Cuál es la diferencia entre transformaciones estrechas (narrow) y amplias (wide) en Spark?

3) ¿Qué es un "Job" en Apache Spark y cómo se ejecuta?

Características Principales de Spark SQL

Compatibilidad con SQL estándar

Permite ejecutar consultas SQL tradicionales y aprovechar funciones avanzadas.

Optimización de consultas

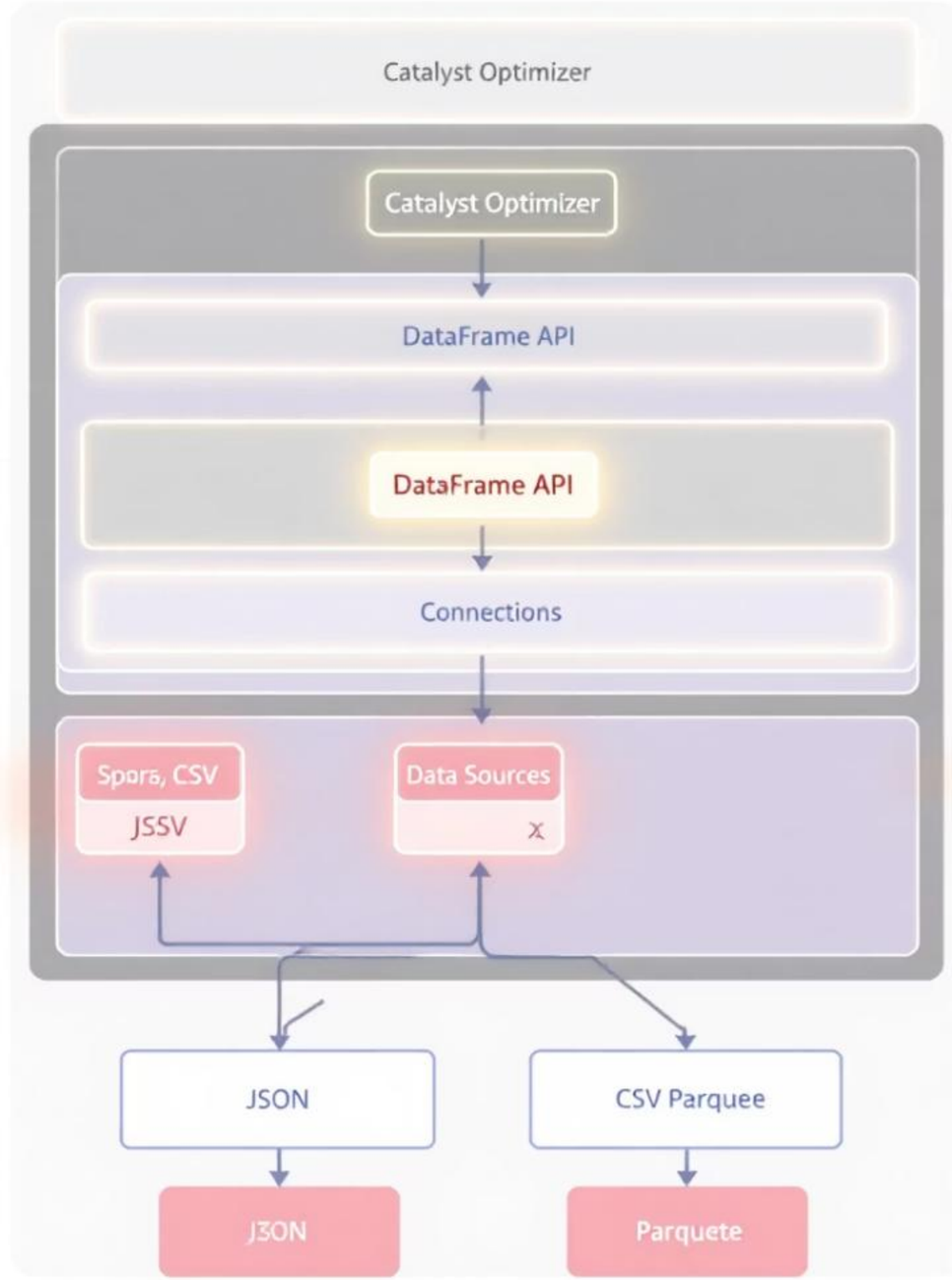
Usa el Catalyst Optimizer, que mejora el rendimiento analizando y transformando consultas.

Integración con DataFrames y Datasets

Se puede trabajar con datos en estructuras optimizadas para un mejor rendimiento.

Soporte para múltiples fuentes

Compatible con archivos Parquet, JSON, ORC, bases de datos relacionales (JDBC), Hive, entre otros.



DataFrames en Spark: Estructura y Creación

Un DataFrame en Spark es una estructura de datos distribuida y optimizada que organiza los datos en un formato tabular similar a una tabla en una base de datos relacional o un DataFrame en Pandas. Se basa en RDDs (Resilient Distributed Datasets), pero con una capa de abstracción adicional que permite realizar operaciones de manera más eficiente y declarativa.

Métodos de Creación

- Desde una lista de Python (Rows o Diccionarios)
- Desde un archivo CSV
- Desde un archivo JSON
- Desde un archivo Parquet
- Desde una Base de Datos (JDBC)

Métodos de Exploración

- show(n) → Muestra las primeras n filas
- printSchema() → Muestra la estructura
- columns → Lista los nombres de columnas
- count() → Devuelve el número total de filas
- describe() → Muestra estadísticas básicas

Creación DataFrame

CREACIÓN DE UN DATAFRAME, Datos como Row objects

```python
from pyspark.sql import SparkSession, Row

# Crear sesión de Spark
spark = SparkSession.builder.appName("CreacionDataFrame").getOrCreate()

# Datos como Row objects
datos = [
    Row(id=1, nombre="Manuel", edad=33),
    Row(id=2, nombre="Sofía", edad=10),
    Row(id=3, nombre="Carlos", edad=28)
]

# Crear DataFrame
df = spark.createDataFrame(datos)

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

```python
from pyspark.sql import SparkSession
import urllib.request

# Crear la sesión de Spark
spark = SparkSession.builder.appName("LeerCSVPublico").getOrCreate()

# URL del dataset público
csv_url = "https://people.sc.fsu.edu/~jburkardt/data/csv/airtravel.csv"

# Ruta local para guardar el archivo temporalmente
ruta_local = "/tmp/airtravel.csv"

# Descargar el archivo CSV
urllib.request.urlretrieve(csv_url, ruta_local)

# Leer el archivo CSV con Spark
df_csv = spark.read.option("header", True).option("inferSchema", True).csv(ruta_local)

# Mostrar los primeros registros
df_csv.show()
```

Consultas y Operaciones con DataFrames

Filtrado

Seleccionar filas que cumplan condiciones específicas

Ordenación

Organizar datos según criterios ascendentes o descendentes

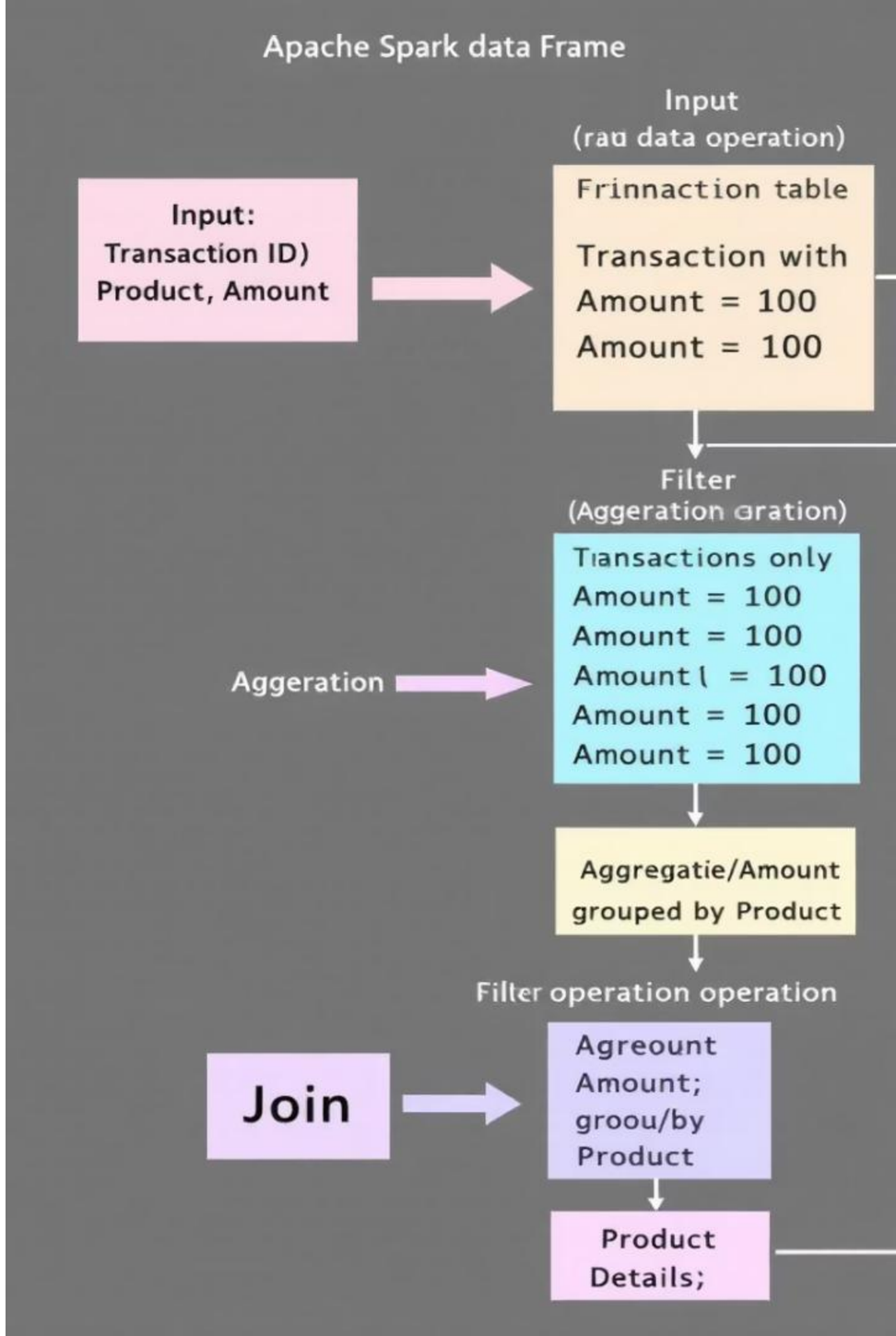
Agregación

Aplicar funciones como SUM(), AVG(), COUNT(), MAX() y MIN()

Unión

Combinar datos de diferentes DataFrames mediante joins

Spark SQL admite una amplia gama de funciones agregadas y analíticas que se pueden utilizar en consultas. Estas operaciones permiten transformar y analizar grandes volúmenes de datos de manera eficiente, aprovechando el procesamiento distribuido de Spark para obtener resultados rápidos incluso con conjuntos de datos masivos.



Consultas y Operaciones con DataFrames

```
# Renombrar columnas para eliminar las comillas dobles
new_columns = [col.strip('"') for col in df_csv.columns]
df_csv = df_csv.toDF(*new_columns)

# Registrar el DataFrame como una vista temporal
df_csv.createOrReplaceTempView("airtravel")

# Consultas utilizando funciones agregadas
spark.sql("SELECT SUM(1958) AS total_1958 FROM airtravel").show()
spark.sql("SELECT AVG(1959) AS promedio_1959 FROM airtravel").show()
spark.sql("SELECT COUNT(1960) AS total_filas_1960 FROM airtravel").show()
spark.sql("SELECT MAX(1958) AS max_1958 FROM airtravel").show()
spark.sql("SELECT MIN(1960) AS min_1960 FROM airtravel").show()
```

Código completo:

https://colab.research.google.com/drive/1lWC6Le1BfgtB7EXdnLB-EcZSNC_k6eX8?usp=sharing

Optimización de Consultas: Catalyst y Tungsten

Catalyst Optimizer

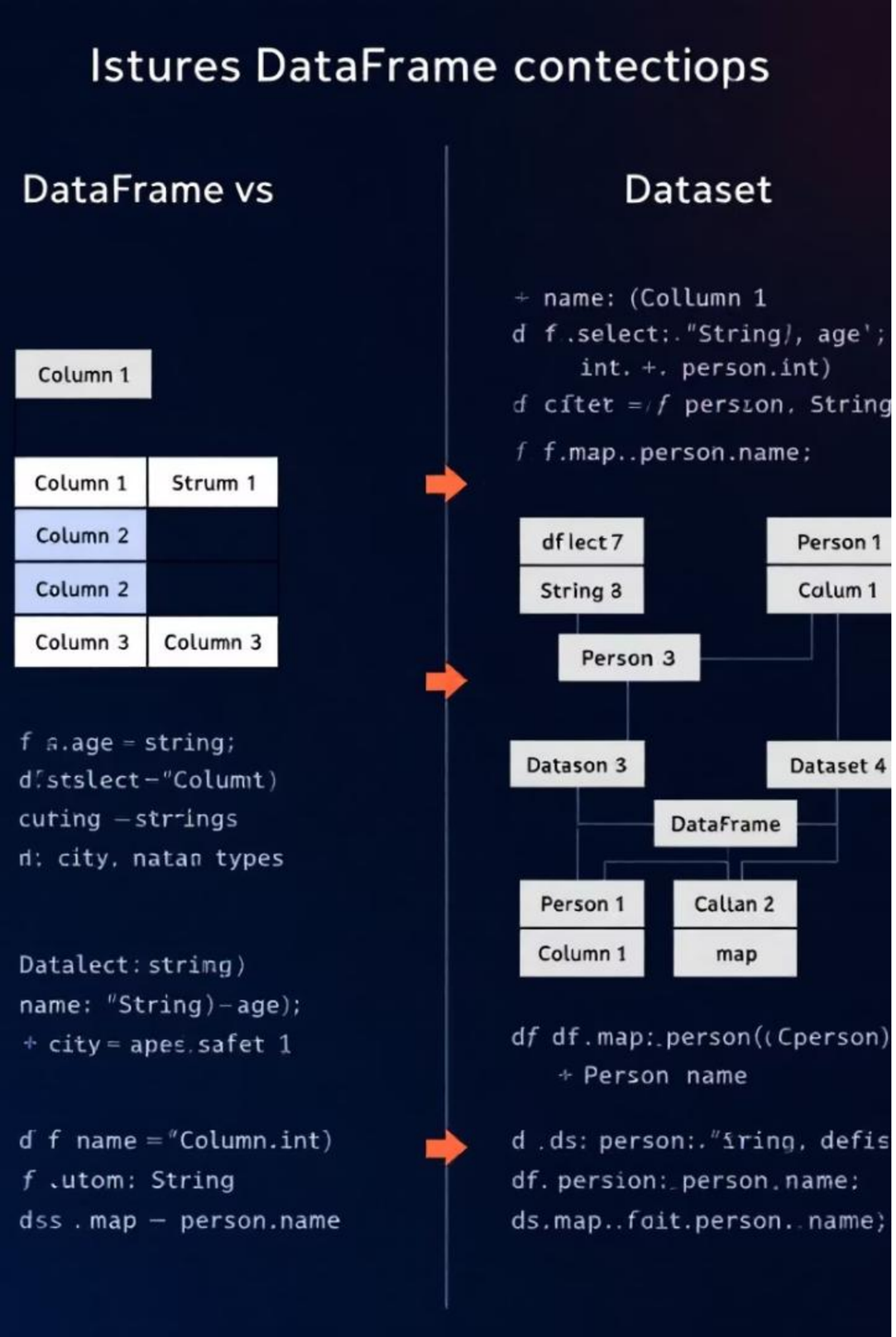
Es el optimizador de consultas de Spark SQL, diseñado para mejorar la eficiencia de las consultas. Basado en un enfoque de optimización por reglas y reescritura de consultas, juega un papel crucial en la optimización antes de la ejecución.

Catalyst transforma las consultas en planes lógicos y físicos optimizados, aplicando reglas como la eliminación de columnas no utilizadas, la combinación de filtros y la optimización de joins.

Proyecto Tungsten

Es un proyecto de optimización de bajo nivel que mejora la ejecución de las consultas en Spark. Se centra principalmente en la gestión eficiente de la memoria, la ejecución de operaciones en paralelo y la optimización de la representación de datos.

Tungsten utiliza técnicas como la generación de código en tiempo de ejecución, la gestión directa de memoria y formatos de datos optimizados para columnas.



DataFrames vs Datasets: Comparativa

Característica	DataFrame	Dataset
Tipado	Débilmente tipado	Fuertemente tipado
Verificación	En tiempo de ejecución	En tiempo de compilación
API	Expresiva y sencilla	Orientada a objetos
Rendimiento	Optimizado para análisis	Equilibrio entre rendimiento y seguridad
Uso ideal	Análisis exploratorio	Aplicaciones empresariales

Un DataFrame es una estructura de datos distribuida de tipo tabular que se utiliza para trabajar con datos estructurados y semiestructurados en Spark, similar a los DataFrames en pandas o R. Por otro lado, un Dataset combina lo mejor de los DataFrames y los RDDs, ofreciendo seguridad de tipos en tiempo de compilación.

Transformar un dataframe a dataset

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import col
from pyspark.sql import Row

# Crear la sesión de Spark
spark = SparkSession.builder.appName("DatasetExample").getOrCreate()

# Definir una clase de caso
Person = Row("name", "age")

# Crear un Dataset a partir de una lista de objetos Row
data = [Person("John", 28), Person("Sarah", 30), Person("Mike", 35)]
df = spark.createDataFrame(data)

# Convertir el DataFrame en un Dataset con el tipo Person
ds = df.rdd.map(lambda row: Person(row["name"], row["age"])).toDF()

# Mostrar el Dataset
ds.show()

# Realizar una operación de filtrado sobre el Dataset
ds.filter(col("age") > 30).show()
```

Funciones Definidas por el Usuario (UDFs)

Definir la función Python

Escribe una función en Python que realice la operación personalizada deseada, como transformaciones de texto, cálculos complejos o lógica de negocio específica.

Registrar la UDF

Registra la función en Spark utilizando el método spark.udf.register() para hacerla disponible en el contexto de Spark SQL.

Aplicar la UDF

Utiliza la función registrada en operaciones sobre DataFrames o en consultas SQL para transformar los datos según tus necesidades específicas.

Las UDFs permiten extender la funcionalidad del motor de procesamiento de datos de Spark con operaciones personalizadas que no están disponibles de manera predeterminada. Son compatibles tanto con SQL como con la API de DataFrame, lo que las hace muy versátiles.

Funciones Definidas por el Usuario (UDFs)

```python
# Definir una función Python
def square(x):
    return x * x

# Registrar la UDF
square_udf = udf(square, IntegerType())

# Aplicar la UDF al DataFrame
df_with_square = df.withColumn("square", square_udf(df["number"]))
```

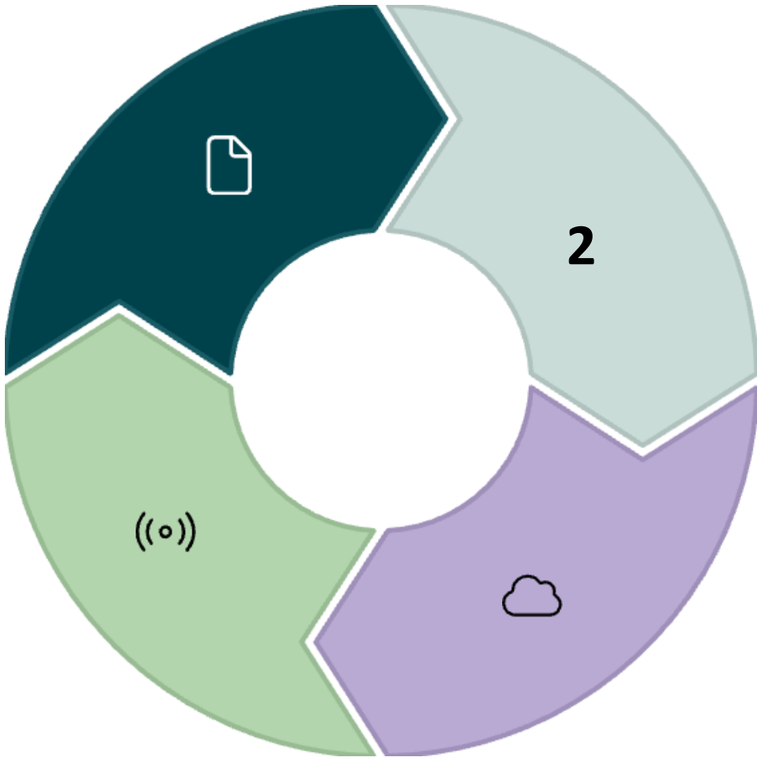
Código completo:

https://colab.research.google.com/drive/1HdNFvCCtp5Q2CkmB9kL6sb2eWOtytyPP?usp=sharing

Fuentes y Destinos de Datos en Spark

Archivos

CSV, JSON, Parquet, ORC, Avro



Bases de Datos

MySQL, PostgreSQL, MongoDB, Cassandra

Streaming

Kafka, Flume, Kinesis

Almacenamiento en Nube

S3, Azure Blob, Google Cloud Storage

Spark ofrece conectores que facilitan la integración con diferentes fuentes y destinos de datos. Estos conectores permiten leer y escribir desde una variedad de sistemas, como bases de datos relacionales (JDBC), MongoDB, Kafka para datos en tiempo real, HDFS, Amazon S3 y Cassandra, entre otros.

Interoperación de los RDD

Spark permite la interoperabilidad entre RDD, DataFrame y Dataset para que los usuarios puedan usar la abstracción que más les convenga según las necesidades de procesamiento y optimización.

- Conversión de RDD a DataFrame
- Conversión de DataFrame a RDD
- Conversión de RDD a Dataset

Ventajas de la Interoperación de los RDDs

- Flexibilidad
- Optimización
- Acceso de bajo.

Formatos de Archivo: JSON vs Parquet

JSON

- Formato basado en texto, legible por humanos
- Estructura flexible y autodescriptiva
- Mayor tamaño de almacenamiento
- Procesamiento más lento
- Ideal para intercambio de datos y APIs

Parquet

- Formato columnar binario optimizado
- Compresión eficiente de datos
- Menor tamaño de almacenamiento
- Procesamiento más rápido
- Ideal para análisis de grandes volúmenes

Casos de Uso

- JSON: Carga de datos de fuentes externas, APIs
- Parquet: Almacenamiento a largo plazo, análisis
- Estrategia común: Ingerir en JSON, transformar y guardar en Parquet

Lectura y Escritura Desde/Hacia Distintas Fuentes

Spark ofrece soporte para leer y escribir archivos en varios formatos, incluidos CSV, JSON, Parquet, ORC, y Avro, entre otros. Estos formatos son comunes cuando se trabaja con datos almacenados en sistemas de archivos distribuidos como HDFS (Hadoop Distributed File System) o en almacenamiento en la nube como Amazon S3 o Azure Blob Storage.

```python
# Leer un archivo CSV
df_csv = spark.read.option("header", "true").option("inferSchema", "true").csv("/path/to/file.csv")
df_csv.show()

# Leer un archivo JSON
df_json = spark.read.json("/path/to/file.json")
df_json.show()

# Leer un archivo Parquet
df_parquet = spark.read.parquet("/path/to/file.parquet")
df_parquet.show()
```

```python
# Escribir un DataFrame en un archivo CSV
df_csv.write.option("header", "true").csv("/path/to/output.csv")

# Escribir un DataFrame en un archivo JSON
df_json.write.json("/path/to/output.json")

# Escribir un DataFrame en un archivo Parquet
df_parquet.write.parquet("/path/to/output.parquet")
```

Actividad Práctica Guiada

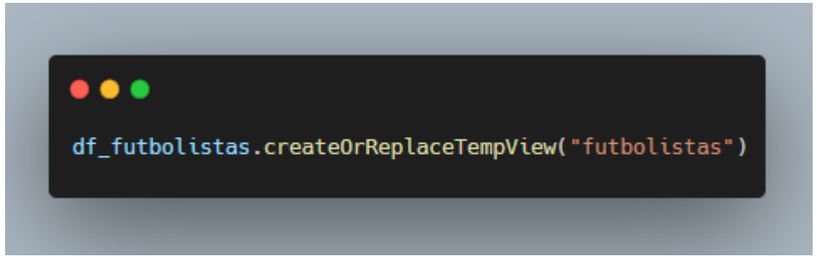
En esta actividad, vamos a utilizar Spark SQL para trabajar con una tabla ficticia que contiene información de futbolistas. Crearemos un DataFrame con datos de futbolistas, como nombre, edad, estatura, rapidez, goles marcados y equipo. Luego, realizaremos algunas consultas básicas usando SQL para obtener información relevante de los datos.

Paso 1: Configuración de Spark y Creación del DataFrame

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import col

# Crear una sesión de Spark
spark = SparkSession.builder.appName("FutbolistasSQL").getOrCreate()

# Crear un DataFrame con datos ficticios de futbolistas
data = [
    ("Lionel Messi", 34, 1.70, 85, 10, "PSG"),
    ("Cristiano Ronaldo", 36, 1.87, 88, 15, "Manchester United"),
    ("Kylian Mbappé", 22, 1.78, 95, 12, "PSG"),
    ("Neymar Jr.", 29, 1.75, 92, 8, "PSG"),
    ("Mohamed Salah", 29, 1.75, 90, 13, "Liverpool"),
    ("Robert Lewandowski", 33, 1.85, 87, 20, "Bayern Munich"),
    ("Kevin De Bruyne", 30, 1.81, 80, 5, "Manchester City")
]

# Definir los nombres de las columnas
columns = ["Nombre", "Edad", "Estatura", "Rapidez", "Goles", "Equipo"]

# Crear el DataFrame
df_futbolistas = spark.createDataFrame(data, columns)

# Mostrar el DataFrame
df_futbolistas.show()
```

Actividad Práctica Guiada

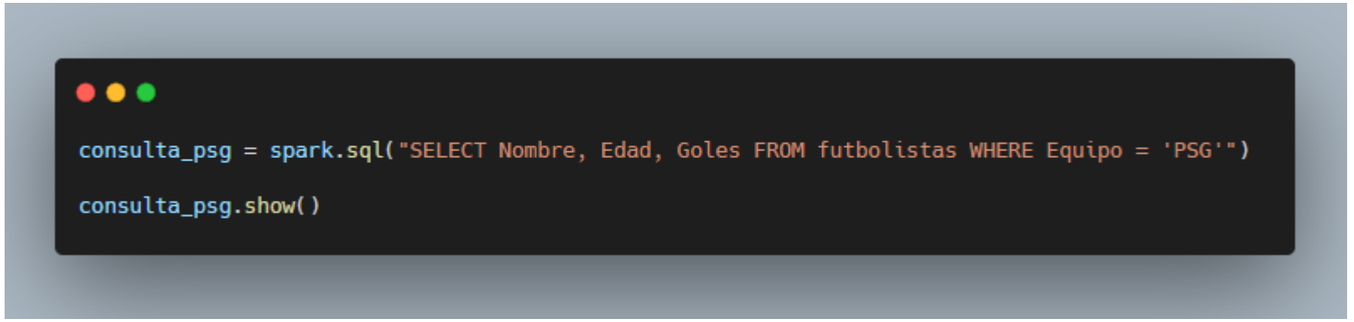
Paso 2: Registrar el DataFrame como una vista temporal para SQL



```
df_futbolistas.createOrReplaceTempView("futbolistas")
```

Paso 3: realizar consultas SQL básicas:

Obtener futbolistas del PSG



```
consulta_psg = spark.sql("SELECT Nombre, Edad, Goles FROM futbolistas WHERE Equipo = 'PSG'")

consulta_psg.show()
```

Actividad Práctica Guiada

- **Obtener futbolistas mayores de 30 años**
- **Calcular el promedio de goles**
- **Obtener el futbolista con mayor rapidez**
- **Obtener el futbolista con mayor estatura**

```
# Obtener futbolistas mayores de 30 años
consulta_mayores_30 = spark.sql("SELECT Nombre, Edad, Goles FROM futbolistas WHERE Edad > 30")
consulta_mayores_30.show()

# Calcular el promedio de goles
promedio_goles = spark.sql("SELECT AVG(Goles) AS Promedio_Goles FROM futbolistas")
promedio_goles.show()

# Obtener el futbolista con mayor rapidez
consulta_futbolista_rapido = spark.sql("SELECT Nombre, Rapidez FROM futbolistas ORDER BY Rapidez DESC
LIMIT 1")
consulta_futbolista_rapido.show()

# Obtener el futbolista con mayor estatura
consulta_futbolista_estatura = spark.sql("SELECT Nombre, Estatura FROM futbolistas ORDER BY Estatura
DESC LIMIT 1")
consulta_futbolista_estatura.show()
```

Código completo:

https://colab.research.google.com/drive/1Vp2mL_DY9OjSFNnIVy109QkH9M4k6xV-?usp=sharing

Reflexión Final

1. ¿Cómo mejora el uso de Catalyst y Tungsten el rendimiento de las consultas en Spark SQL, y por qué son importantes para el procesamiento de Big Data?

2. ¿En qué casos sería más conveniente utilizar un DataFrame sobre un Dataset en Apache Spark?

3. ¿Cómo contribuyen las Funciones Definidas por el Usuario (UDFs) en Apache Spark a la personalización de operaciones y qué consideraciones deben tenerse en cuenta al utilizarlas?