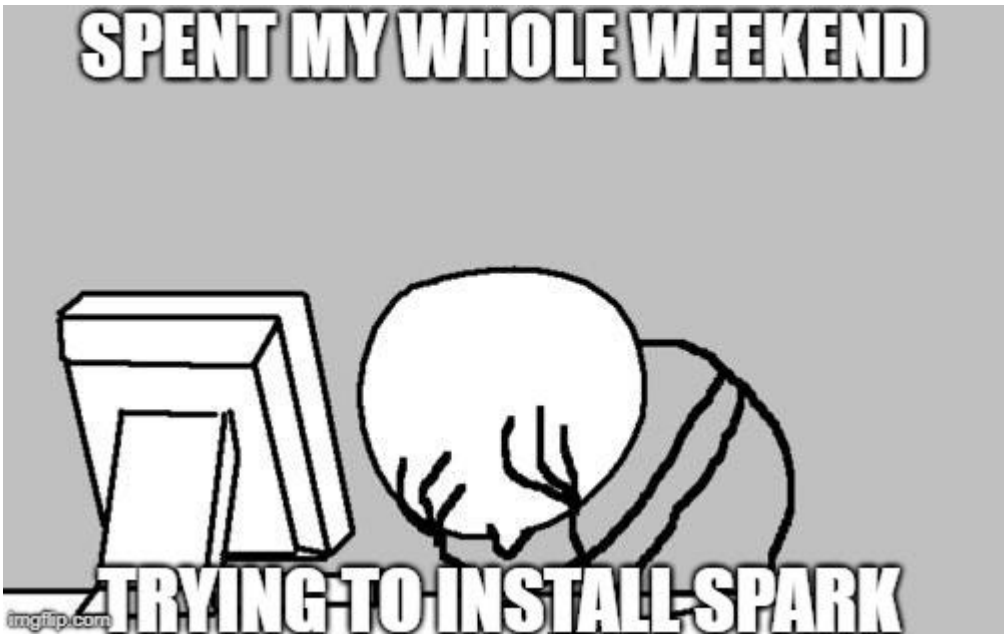
PySpark SQL



¿Qué es PySpark SQL?

PySpark SQL es un módulo de Apache Spark que proporciona una interfaz de programación en Python para el procesamiento de datos estructurados y semiestructurados. Combina la potencia de Spark con la facilidad de uso de Python, permitiendo a los desarrolladores realizar análisis de datos y consultas utilizando SQL y operaciones de manipulación de datos en un entorno distribuido y de alto rendimiento.

Ventajas de utilizar PySpark SQL

- **Escalabilidad**: PySpark SQL aprovecha la capacidad de procesamiento distribuido de Apache Spark, lo que permite el procesamiento eficiente de grandes volúmenes de datos en clústeres de múltiples nodos.
- **Facilidad de uso**: PySpark SQL proporciona una API en Python que es más accesible para los desarrolladores que ya están familiarizados con el lenguaje.
- **Soporte para SQL**: Permite a los usuarios ejecutar consultas SQL sobre sus datos, lo que facilita la realización de análisis y agregaciones complejas.
- **Optimización de consultas**: PySpark SQL realiza optimizaciones internas para acelerar el procesamiento de datos, como la ejecución de operaciones en memoria y la planificación de consultas eficiente.
- **Integración con otras fuentes de datos**: PySpark SQL se integra con una variedad de fuentes de datos, incluidos sistemas de almacenamiento en la nube y bases de datos externas.
- **Procesamiento en tiempo real**: PySpark SQL es compatible con Structured Streaming, lo que permite el procesamiento de datos en tiempo real.

Conceptos básicos

Notaciones

- `[]` (Operaciones de Columnas Individuales): Utilizas df[nombre_de_columna] para acceder a una columna específica en un DataFrame df. Puedes realizar operaciones y transformaciones en una sola columna utilizando esta notación.

```
# Acceder a la columna "Edad" del DataFrame df
df["Edad"]

# Aplicar una transformación a la columna "Edad"
df["Edad"] * 2
```

- `df[]` (Filtrado de Filas por Condición): Utilizas df[condición] para filtrar las filas de un DataFrame df basándote en una condición dada. Esto devuelve un nuevo DataFrame con las filas que cumplen con la condición

```
# Filtrar las filas donde la columna "Edad" es mayor que 30
df[df["Edad"] > 30]
```

- `df[df[]]` (Filtrado de Filas y Selección de Columnas): Puedes combinar las dos notaciones para filtrar filas y seleccionar columnas específicas en un DataFrame. Primero, aplicas un filtro para seleccionar las filas deseadas y luego utilizas [] para seleccionar las columnas de interés.

```
# Filtrar las filas donde "Edad" es mayor que 30 y seleccionar solo las
columnas "Nombre" y "Edad"
df[df["Edad"] > 30]["Nombre", "Edad"]
```

Dataframe y Dataset ¿Qué son y en que se diferencian?

DataFrames: Los DataFrames en PySpark SQL son estructuras de datos tabulares, similares a las tablas de una base de datos o a un DataFrame en Pandas. Cada columna tiene un nombre y un tipo de datos asociado. Los DataFrames son inmutables y distribuidos, lo que los hace adecuados para el procesamiento paralelo de datos.

```
# Importar la biblioteca PySpark
from pyspark.sql import SparkSession

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

# Crear un DataFrame a partir de una lista de tuplas
data = [("Alice", 34), ("Bob", 45), ("Charlie", 29)]
columnas = ["Nombre", "Edad"]
df = spark.createDataFrame(data, columns)
```

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

Datasets: Los Datasets son una extensión de los DataFrames que proporcionan tipado estático y funcionalidades orientadas a objetos. Los Datasets son más eficientes en cuanto a rendimiento que los DataFrames estándar cuando se utilizan con funciones lambda y permiten a los desarrolladores aprovechar las ventajas de la programación funcional.

```
# Importar la biblioteca PySpark
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType, StructField, StringType, IntegerType

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

# Definir el esquema del Dataset
schema = StructType([
    StructField("Nombre", StringType(), True),
    StructField("Edad", IntegerType(), True)
])

# Crear un Dataset a partir de una lista de tuplas y aplicar el esquema
data = [("Alice", 34), ("Bob", 45), ("Charlie", 29)]
df = spark.createDataFrame(data, schema=schema)

# Convertir el DataFrame en un Dataset
dataset = df.as[("Nombre", "Edad")]

# Mostrar el Dataset
dataset.show()
```

RDDs

Resilient Distributed Datasets (RDDs): Los RDDs son la estructura de datos fundamental en Spark. Son colecciones inmutables y distribuidas de objetos que se pueden procesar de manera paralela. Aunque los DataFrames y Datasets son más convenientes para el procesamiento de datos estructurados, los RDDs son más flexibles y se utilizan en situaciones donde se requiere mayor control sobre la manipulación de datos.

```
# Importar la biblioteca PySpark
from pyspark import SparkContext

# Crear un contexto de Spark
sc = SparkContext("local", "EjemploRDD")

# Crear un RDD a partir de una lista
data = [1, 2, 3, 4, 5]
rdd = sc.parallelize(data)
```

```
# Realizar una operación de mapeo en el RDD
rdd_mapeado = rdd.map(lambda x: x * 2)

# Mostrar los elementos del RDD resultante
print(rdd_mapeado.collect())

# Detener el contexto de Spark
sc.stop()
```

Operaciones básicas con DataFrames

Crear DataFrame

Puedes crear un DataFrame a partir de diferentes fuentes de datos, como listas, diccionarios o archivos.

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("EjemploDataFrame").getOrCreate()

data = [("Alice", 34), ("Bob", 45), ("Charlie", 29)]
columnas = ["Nombre", "Edad"]

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

Mostrar el contenido de un DataFrame

Puedes usar el método show() para ver los primeros n registros del DataFrame.

```
df.show()
```

Filtrado de datos

Puedes filtrar filas de un DataFrame según una condición.

```
df.filter(df["Edad"] > 30).show()
```

Agregaciones

Puedes realizar operaciones de agregación en un DataFrame, como contar, sumar o calcular estadísticas.

```
df.groupBy("Edad").count().show()
df.groupBy("Nombre").agg({"Edad": "avg"}).show()
```

Ordenar

Puedes ordenar las filas de un DataFrame por una o varias columnas.

```
df.orderBy("Edad").show()
```

Tipos de datos

Tipos Primitivos:	Tipos Compuestos:	Tipos de Datos de Fecha y Hora:	Tipos de Datos Binarios:	Tipos de Datos Nulos:
IntegerType	ArrayType	DateType	BinaryType	NullType
StringType	MapType	TimestampType		
DoubleType	StructType			
BooleanType	StructField			
LongType				
ShortType				
FloatType				
ByteType				

Operaciones con columnas

Agregar nuevas columnas

Puedes agregar nuevas columnas basadas en operaciones en columnas existentes.

```
from pyspark.sql.functions import col

df.withColumn("DobleEdad", col("Edad") * 2).show()
```

Eliminar columnas

Puedes eliminar una o varias columnas de un DataFrame.

```
df.drop("Edad").show()
```

Selección de columnas

Puedes seleccionar una o varias columnas de un DataFrame.

```
df.select("Nombre").show()
df.select("Nombre", "Edad").show()
```

Renombrar columnas

```
df.withColumnRenamed("Edad", "Años").show()
```

Funciones integradas

Función	Descripción	Ejemplo de Uso
`count`	Contar el número de filas	`df.count()`
`avg`	Calcular el promedio de una columna	`df.agg({"Edad": "avg"}).show()`
`sum`	Calcular la suma de una columna	`df.agg({"Edad": "sum"}).show()`
`min`	Encontrar el valor mínimo de una columna	`df.agg({"Edad": "min"}).show()`
`max`	Encontrar el valor máximo de una columna	`df.agg({"Edad": "max"}).show()`
`sqrt`	Calcular la raíz cuadrada de una columna	`df.withColumn("RaizEdad", sqrt(col("Edad"))).show()`
`abs`	Calcular el valor absoluto de una columna	`df.withColumn("AbsEdad", abs(col("Edad"))).show()`
`round`	Redondear los valores de una columna	`df.withColumn("EdadRedondeada", round(col("Edad"), 2)).show()`
`concat`	Concatenar valores de columnas	`df.withColumn("NombreCompleto", concat(col("Nombre"), lit(" "), col("Apellido"))).show()`

Función	Descripción	Ejemplo de Uso
`substring`	Extraer una subcadena de una columna	`df.withColumn("SubNombre", substring(col("Nombre"), 1, 3)).show()`
`date_format`	Formatear fechas	`df.withColumn("FechaFormateada", date_format(col("Fecha"), "dd-MM-yyyy")).show()`
`when`	Evaluar condiciones y aplicar resultados	`df.withColumn("Categoría", when(col("Edad") > 30, "Mayor").otherwise("Menor")).show()`

Consultas

Cargar y guardar datos

- **Cargar Datos desde Archivos**: Cómo cargar datos desde archivos locales o remotos en un DataFrame de PySpark

```
df = spark.read.csv("datos.csv", header=True, inferSchema=True)
```

- **Guardar Datos en Formatos Diversos**: Cómo guardar los resultados de tu análisis en diferentes formatos, como Parquet, CSV, JSON, etc

```
# Guardar el DataFrame en formato Parquet
df.write.parquet("datos.parquet")

# Guardar el DataFrame en formato JSON
df.write.json("datos.json")
```

- **Conexión a Bases de Datos Externas**: Cómo conectarse y cargar datos desde bases de datos externas como PostgreSQL, MySQL o SQL Server

```
# Conexión a una base de datos MySQL
df_mysql = spark.read.format("jdbc").option("url",
"jdbc:mysql://localhost:3306/mi_db").option("dbtable", "mi_tabla").option("user",
"usuario").option("password", "contraseña").load()

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

Consultas SQL

- **Ejecución de Consultas SQL**: Cómo ejecutar consultas SQL en PySpark SQL utilizando el método sql().

```

```

```
    # Registrar el DataFrame como una tabla temporal
    df.createOrReplaceTempView("mi_tabla")

    # Ejecutar una consulta SQL
    resultados = spark.sql("SELECT Nombre, Edad FROM mi_tabla WHERE Edad > 30")

    # Mostrar los resultados
    resultados.show()
```

- **Consulta de Múltiples Fuentes**: Cómo realizar consultas que involucran múltiples DataFrames y tablas.

```
    # Cargar otro DataFrame desde un archivo
    df2 = spark.read.csv("otros_datos.csv", header=True, inferSchema=True)
    df2.createOrReplaceTempView("otros_datos")

    # Realizar una consulta que involucre ambos DataFrames
    consulta_compleja = spark.sql("""
        SELECT t1.Nombre, t1.Edad, t2.OtroCampo
        FROM mi_tabla t1
        JOIN otros_datos t2
        ON t1.Nombre = t2.Nombre
    """)
    consulta_compleja.show()
```

- **Funciones SQL Integradas**: Uso de funciones SQL integradas en tus consultas.

```
    from pyspark.sql.functions import expr

    # Utilizar funciones SQL integradas en una consulta
    consulta_con_funciones = df.selectExpr("Nombre", "Edad", "sqrt(Edad) as
RaizEdad")
    consulta_con_funciones.show()
```

Operaciones de filtrado y selección

- **Filtrado de Filas**: Cómo filtrar filas de un DataFrame basado en condiciones específicas utilizando métodos como filter() o SQL WHERE.

```
    # Filtrar filas basadas en una condición
    df_filtrado = df.filter(df["Edad"] > 30)
    df_filtrado.show()
```

- **Selección de Columnas**: Cómo seleccionar columnas específicas de un DataFrame utilizando select() o SQL.

```
# Seleccionar columnas específicas
df_seleccionado = df.select("Nombre", "Edad")
df_seleccionado.show()
```

- **Operaciones de Transformación de Columnas**: Cómo aplicar operaciones de transformación a columnas utilizando métodos como withColumn().

```
from pyspark.sql.functions import col

# Agregar una nueva columna calculada
df_transformado = df.withColumn("DobleEdad", col("Edad") * 2)
df_transformado.show()
```

- **Ordenar y Agregar Columnas Calculadas**: Cómo ordenar las filas de un DataFrame y agregar nuevas columnas calculadas.

```
# Ordenar las filas por la columna Edad y agregar una nueva columna calculada
df_ordenado = df.orderBy("Edad")
df_ordenado = df_ordenado.withColumn("EdadDuplicada", col("Edad") * 2)
df_ordenado.show()
```

Agrupación y agregación de datos

- **GroupBy y Agregación Básica**: La función groupBy se utiliza para agrupar filas por una o varias columnas y luego puedes aplicar funciones de agregación, como sum, count, avg, etc., en esas agrupaciones.

```
# Agrupar por la columna "Ciudad" y calcular la suma de las ventas para cada
ciudad
df.groupBy("Ciudad").agg({"Ventas": "sum"}).show()
```

- **Agregación con Alias**: Puedes asignar nombres de columna más significativos a los resultados de las operaciones de agregación utilizando la función alias.

```
from pyspark.sql.functions import sum

# Calcular la suma de las ventas y asignar un nombre a la columna resultante
df.groupBy("Ciudad").agg(sum("Ventas").alias("TotalVentas")).show()
```

- **Uso de la función** `agg`: La función agg permite realizar múltiples operaciones de agregación en un solo paso.

```
    from pyspark.sql.functions import sum, avg, max

    # Calcular la suma, el promedio y el máximo de las ventas para cada ciudad
    df.groupBy("Ciudad").agg(sum("Ventas").alias("TotalVentas"),
avg("Ventas").alias("PromedioVentas"), max("Ventas").alias("MaxVentas")).show()
```

- **Uso de expresiones SQL**: Puedes utilizar expresiones SQL para realizar operaciones de agregación más complejas

```
    # Calcular la suma de las ventas y el promedio de las ganancias para cada
ciudad
    df.createOrReplaceTempView("tabla_ventas")
    resultados = spark.sql("SELECT Ciudad, SUM(Ventas) AS TotalVentas,
AVG(Ganancias) AS PromedioGanancias FROM tabla_ventas GROUP BY Ciudad")
    resultados.show()
```

Unión y combinación de DataFrames

- **Unión (join)**: La unión combina dos DataFrames en función de una columna común o una clave.

```
    # Unir dos DataFrames en función de la columna "ID"
    df1 = spark.createDataFrame([(1, "Alice"), (2, "Bob")], ["ID", "Nombre"])
    df2 = spark.createDataFrame([(2, "Ventas"), (3, "Marketing")], ["ID",
"Departamento"])

    resultado = df1.join(df2, "ID", "inner")
    resultado.show()
```

- **Combinación (union)**: La combinación agrega las filas de un DataFrame a otro. Ambos DataFrames deben tener la misma estructura.

```
    # Combinar dos DataFrames
    df1 = spark.createDataFrame([(1, "Alice"), (2, "Bob")], ["ID", "Nombre"])
    df2 = spark.createDataFrame([(3, "Charlie"), (4, "David")], ["ID", "Nombre"])

    resultado = df1.union(df2)
    resultado.show()
```

- **Unión externa (outer join)**: La unión externa combina dos DataFrames y conserva todas las filas de ambos DataFrames, llenando los valores faltantes con nulos.

```
    # Unión externa para conservar todas las filas de ambos DataFrames
    df1 = spark.createDataFrame([(1, "Alice"), (2, "Bob")], ["ID", "Nombre"])
    df2 = spark.createDataFrame([(2, "Ventas"), (3, "Marketing")], ["ID",
"Departamento"])

    resultado = df1.join(df2, "ID", "outer")
    resultado.show()
```

- **Unión izquierda (left join) y unión derecha (right join)**: Las uniones izquierda y derecha conservan todas las filas del DataFrame izquierdo o derecho, respectivamente, y llenan con nulos las filas del otro DataFrame que no tienen coincidencias.

```
    # Unión izquierda (left join)
    resultado_izquierda = df1.join(df2, "ID", "left")
    resultado_izquierda.show()

    # Unión derecha (right join)
    resultado_derecha = df1.join(df2, "ID", "right")
    resultado_derecha.show()
```

Ventanas en PySpark SQL

En PySpark SQL, las ventanas (windows) permiten realizar cálculos basados en ventanas de filas o grupos de filas dentro de un DataFrame. Esto es útil para calcular valores acumulativos o realizar análisis comparativos.

- **Funciones de Ventana Básicas**: Puedes utilizar funciones de ventana para realizar cálculos sobre un conjunto de filas definido por una ventana. Algunas funciones de ventana comunes incluyen over, partitionBy, orderBy, rowsBetween, rangeBetween, entre otras.

```
    from pyspark.sql import Window
    from pyspark.sql.functions import sum

    # Definir una ventana que particiona por "Departamento" y ordena por "Ventas"
    ventana = Window.partitionBy("Departamento").orderBy("Ventas")

    # Calcular la suma acumulativa de "Ventas" dentro de cada partición
    df.withColumn("SumaAcumulativa", sum("Ventas").over(ventana)).show()
```

- **Ventana sin Particionar**: Puedes usar una ventana sin particionar para realizar cálculos en todo el DataFrame sin tener en cuenta una columna específica.

```
from pyspark.sql import Window
from pyspark.sql.functions import row_number

# Definir una ventana sin particionar y ordenar por "Edad" en orden ascendente
ventana = Window.orderBy("Edad")

# Asignar un número de fila único a cada fila del DataFrame
df.withColumn("NumeroFila", row_number().over(ventana)).show()
```

- **Ventana de Filas y Rangos**: Puedes definir ventanas basadas en un rango específico de filas, lo que te permite realizar cálculos basados en un número fijo de filas antes y después de la fila actual.

```
from pyspark.sql import Window
from pyspark.sql.functions import lag, lead

# Definir una ventana de 2 filas antes y 2 filas después de la fila actual
ventana = Window.orderBy("Fecha").rowsBetween(-2, 2)

# Calcular el valor anterior y siguiente para cada fila en función de la fecha
df.withColumn("ValorAnterior", lag("Valor", 2).over(ventana))
df.withColumn("ValorSiguiente", lead("Valor", 2).over(ventana))
```

- **Funciones de Ventana Avanzadas**: PySpark SQL proporciona una amplia gama de funciones de ventana avanzadas, como rank, dense_rank, ntile, first_value, last_value, entre otras, que se pueden utilizar en combinación con ventanas para realizar cálculos más complejos.

```
from pyspark.sql import Window
from pyspark.sql.functions import rank

# Definir una ventana que particiona por "Departamento" y ordena por "Ventas"
ventana = Window.partitionBy("Departamento").orderBy("Ventas")

# Calcular la clasificación de las ventas dentro de cada partición
df.withColumn("Clasificacion", rank().over(ventana)).show()
```

Estrategias de optimización

En PySpark SQL, las estrategias de optimización son técnicas utilizadas internamente para acelerar el procesamiento de datos y mejorar el rendimiento de las consultas.

- **Pruning de Columnas (Column Pruning)**: La poda de columnas implica eliminar las columnas que no son necesarias para una consulta, lo que reduce la cantidad de datos que se deben cargar y procesar.

```
from pyspark.sql import Window
from pyspark.sql.functions import row_number
```

```
    # Definir una ventana sin particionar y ordenar por "Edad" en orden ascendente
    ventana = Window.orderBy("Edad")

    # Asignar un número de fila único a cada fila del DataFrame
    df.withColumn("NumeroFila", row_number().over(ventana)).show()
```

- **Predicados Pushdown**: Los predicados pushdown permiten llevar a cabo la filtración de datos lo más cerca posible de la fuente de datos, reduciendo así la cantidad de datos que se transfieren a través de la red.

```
    # Aplicar el filtro directamente a la fuente de datos (por ejemplo, una base
de datos)
    df = spark.read.jdbc(url="jdbc:mysql://localhost:3306/mi_base_de_datos",
table="mi_tabla", predicates=["Edad > 30"])
    df.show()
```

- **Optimización de Join (Join Optimization)**: PySpark SQL utiliza diversas estrategias para optimizar las operaciones de unión, como la selección del algoritmo de unión más eficiente según el tamaño de los DataFrames involucrados.

```
    # Unir DataFrames utilizando una clave común
    resultado = df1.join(df2, "ID", "inner")
    resultado.show()
```

- **Almacenamiento en Caché (Caching)**: Puedes almacenar en caché un DataFrame o una vista temporal para evitar recálculos costosos en consultas posteriores.

```
    # Almacenar en caché un DataFrame
    df.cache()
    # Realizar consultas sobre el DataFrame en caché
    df.filter(df["Edad"] > 30).groupBy("Ciudad").count().show()
```

- **Optimización de Consulta (Query Optimization)**: PySpark SQL utiliza un optimizador de consultas para analizar y reorganizar las consultas para mejorar el rendimiento.

```
    # Utilizar funciones de ventana para calcular resultados complejos de manera
eficiente
    from pyspark.sql import Window
    from pyspark.sql.functions import sum

    ventana = Window.partitionBy("Departamento").orderBy("Ventas")
    df.withColumn("SumaAcumulativa", sum("Ventas").over(ventana)).show()
```

Particionamiento y almacenamiento de los datos

Uso de índices y caché

Optimización de consultas

Integración con fuentes de datos externas

La integración con fuentes de datos externas es esencial en el procesamiento de datos con PySpark. Permite a PySpark conectarse a diferentes tipos de fuentes de datos, como bases de datos externas, sistemas de almacenamiento en la nube y archivos en varios formatos.

Conexión con bases de datos externas

PySpark admite la conexión con diversas bases de datos externas, como MySQL, PostgreSQL, Oracle, SQL Server, y más. Para conectarte a una base de datos externa, debes especificar la URL de conexión, credenciales y, opcionalmente, el nombre de la tabla que deseas cargar en un DataFrame.

```
    from pyspark.sql import SparkSession

    spark = SparkSession.builder.appName("EjemploBDExterna").getOrCreate()

    url = "jdbc:mysql://hostname:port/database"
    properties = {
        "user": "username",
        "password": "password",
        "driver": "com.mysql.jdbc.Driver"
    }

    df = spark.read.jdbc(url, "nombre_de_tabla", properties=properties)
```

Integración con fuentes de datos en la nube

PySpark puede conectarse a fuentes de datos en la nube, como Amazon S3 o Azure Blob Storage, para leer y escribir datos. Esto es útil cuando tus datos se almacenan en entornos de nube.

```
    from pyspark.sql import SparkSession

    spark = SparkSession.builder.appName("EjemploS3").getOrCreate()

    df = spark.read.option("header",
"true").csv("s3a://bucket_name/path/to/data.csv")
```

Lectura y escritura de datos en formatos diversos

PySpark es versátil en cuanto a los formatos de datos que puede leer y escribir. Puede leer y escribir datos en formatos como CSV, Parquet, Avro, ORC, JSON, y más.

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("EjemploCSV").getOrCreate()

df = spark.read.option("header", "true").csv("archivo.csv")
```

Buenas practicas

Las "buenas prácticas" en el contexto de PySpark se refieren a las estrategias y pautas recomendadas para el desarrollo, el diseño y la administración de aplicaciones y procesos de PySpark de manera eficiente y efectiva. Algunas buenas prácticas incluyen:

- **Optimización de Código**: Es importante escribir código eficiente y aprovechar al máximo las operaciones en paralelo que ofrece PySpark.
- **Gestión de Memoria**: Monitorizar y gestionar adecuadamente el uso de la memoria para evitar problemas de rendimiento.
- **Uso de Almacenamiento en Caché**: Almacenar en caché DataFrames intermedios cuando sea posible para evitar retrasos en el cálculo.
- **Optimización de Join**: Elegir los algoritmos y tipos de join adecuados para minimizar el costo computacional.
- **Tuning de Configuración**: Ajustar las configuraciones de PySpark según las necesidades específicas de tu aplicación.

Seguridad

La "seguridad" en PySpark se refiere a la implementación de medidas para proteger tus datos, aplicaciones y recursos de cualquier amenaza o acceso no autorizado. Algunas consideraciones de seguridad incluyen:

- **Control de Acceso**: Limitar el acceso a tus recursos de PySpark a usuarios autorizados y roles específicos.
- **Encriptación**: Utilizar encriptación para proteger los datos en tránsito y en reposo.
- **Autenticación y Autorización**: Implementar autenticación de usuarios y definir políticas de autorización para controlar el acceso a las operaciones de PySpark.
- **Auditoría y Registro**: Registrar actividades y eventos de PySpark para el seguimiento y la auditoría.