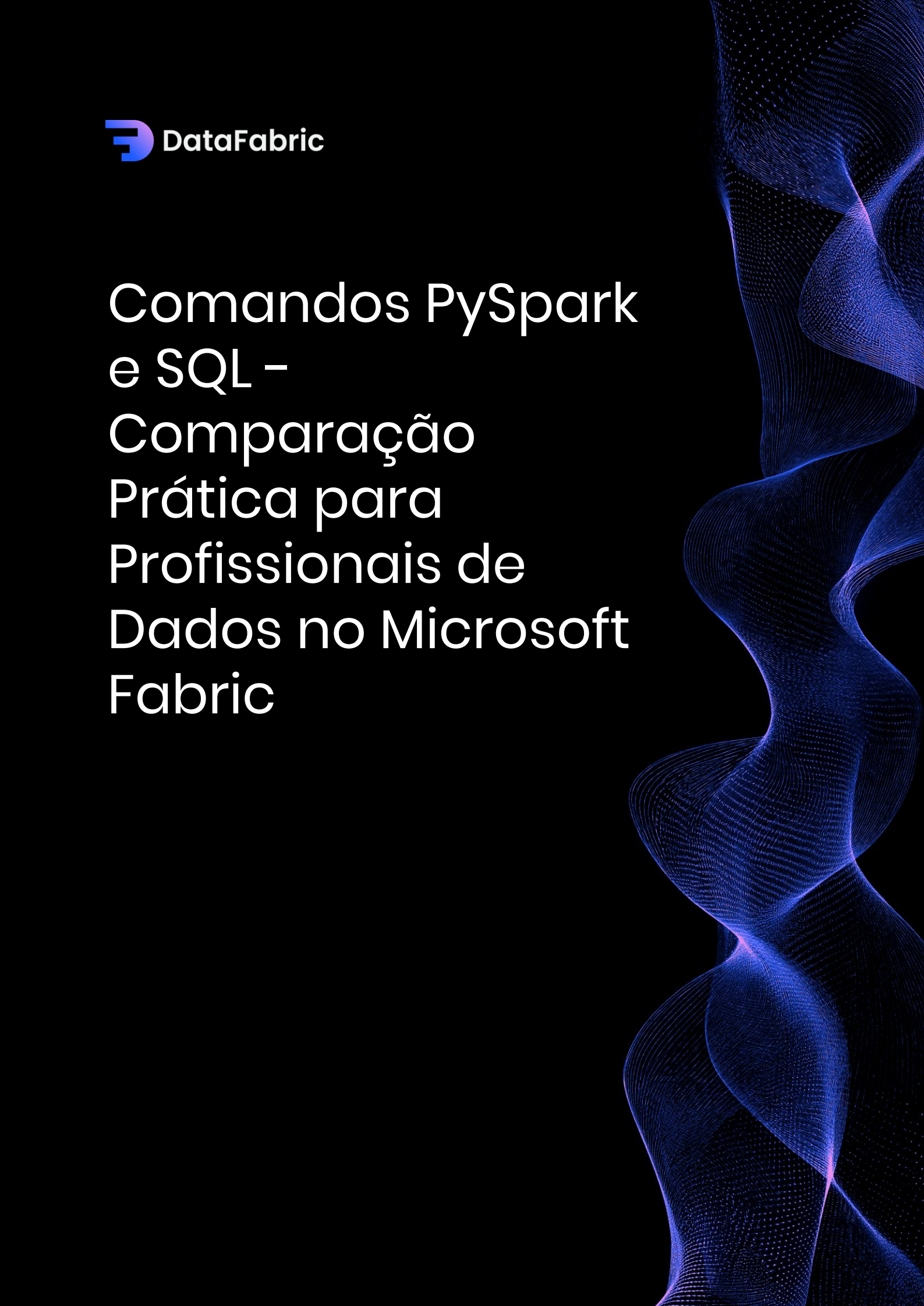
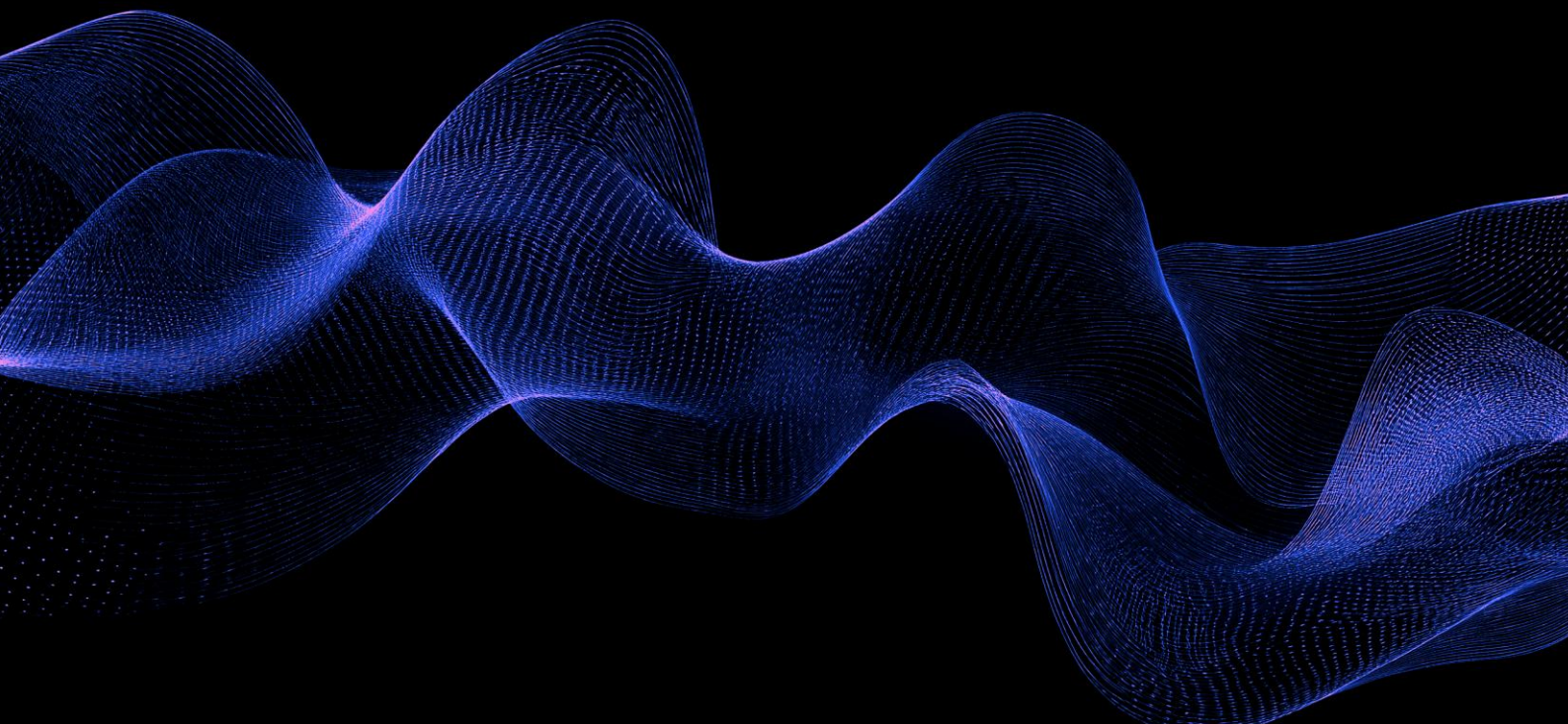
DataFabric

Comandos PySpark e SQL - Comparação Prática para Profissionais de Dados no Microsoft Fabric



Capítulos



Introdução

O Microsoft Fabric oferece um ambiente unificado de análise de dados, onde podemos utilizar tanto SQL quanto PySpark para trabalhar com grandes volumes de dados. Este eBook foi criado para ajudar profissionais que já dominam SQL a entender como os mesmos comandos podem ser traduzidos para PySpark, uma poderosa biblioteca do Apache Spark usada para análise de dados em larga escala.

Aqui, faremos uma comparação lado a lado entre SQL e PySpark, cobrindo desde operações simples de filtragem de dados até agregações e transformações avançadas, com exemplos práticos que podem ser aplicados no Microsoft Fabric.

Introdução ao Ambiente de PySpark no Microsoft Fabric

Antes de mergulharmos nos comandos, é importante entender como configurar um ambiente PySpark no Microsoft Fabric. Este capítulo oferece uma breve introdução sobre como conectar-se ao Fabric, criar notebooks e iniciar seu primeiro script PySpark.Colaboração em Equipe

Criação de um DataFrame com PySpark

```
from pyspark.sql import SparkSession

# Cria uma Spark session
spark = SparkSession.builder.appName("FabricExample").getOrCreate()

# Carrega dados de um arquivo CSV
df = spark.read.csv("/path/to/file.csv", header=True, inferSchema=True)

# Exibe as primeiras linhas do DataFrame
df.show()
```

#1- Seleção de Colunas

Ambos os comandos selecionam colunas específicas de uma tabela ou DataFrame. No PySpark, usamos o método select() para escolher as colunas que queremos visualizar.

SQL

```sql
SELECT coluna1, coluna2 FROM tabela;
```

PySpark

```python
df.select("coluna1", "coluna2").show()
```

2 - Filtragem de Dados

Tanto em SQL quanto em PySpark, podemos filtrar linhas com base em uma condição. No PySpark, o método filter() aplica a condição no DataFrame.

SQL

```sql
SELECT * FROM tabela WHERE coluna1 = 'valor';
```

PySpark

```python
df.filter(df.coluna1 == 'valor').show()
```

3 - Ordenação de Dados

Tanto em SQL quanto em PySpark, podemos filtrar linhas com base em uma condição. No PySpark, o método filter() aplica a condição no DataFrame.

SQL

```sql
SELECT * FROM tabela ORDER BY coluna1 DESC;
```

PySpark

```python
df.orderBy(df.coluna1.desc()).show()
```

4 - Agrupamento e Agregação

O agrupamento é um processo comum em SQL. Em PySpark, o método groupBy() é combinado com funções de agregação como count() para realizar a operação.

SQL

```sql
SELECT coluna1, COUNT(*) FROM tabela GROUP BY coluna1;
```

PySpark

```python
df.groupBy("coluna1").count().show()
```

5 - Junções de Tabelas

Em ambos os casos, a junção de tabelas é feita com base em uma coluna comum. No PySpark, usamos o método join() para realizar essa operação, especificando o tipo de junção (neste caso, inner).

SQL

```sql
SELECT * FROM tabela1 INNER JOIN tabela2 ON tabela1.coluna = tabela2.coluna;
```

PySpark

```python
df1.join(df2, df1.coluna == df2.coluna, "inner").show()
```

6 - Limitação de Linhas

A limitação de resultados funciona de forma semelhante em SQL e PySpark, usando LIMIT em SQL e limit() no PySpark.

SQL

PySpark

7 - Criação de Novas Colunas

A criação de novas colunas, seja por operações matemáticas ou concatenando valores, é feita com o withColumn() no PySpark.

SQL

```sql
SELECT coluna1, coluna2, (coluna1 + coluna2) AS nova_coluna
FROM tabela;
```

PySpark

```python
df.withColumn("nova_coluna", df.coluna1 + df.coluna2).show()
```

8 - Remoção de Duplicatas

A remoção de duplicatas de uma coluna é uma operação comum tanto em SQL quanto no PySpark. No PySpark, o método distinct() executa essa tarefa.

SQL

```sql
SELECT DISTINCT coluna1 FROM tabela;
```

PySpark

```python
df.select("coluna1").distinct().show()
```

9 - Subconsultas e Consultas Complexas

A remoção de duplicatas de uma coluna é uma operação comum tanto em SQL quanto no PySpark. No PySpark, o método distinct() executa essa tarefa.

SQL

```sql
SELECT *
FROM (SELECT * FROM tabela WHERE coluna1 = 'valor') AS subconsulta;
```

PySpark

```python
subconsulta = df.filter(df.coluna1 == 'valor')
subconsulta.show()
```

10 - Salvando Dados em Formatos Diferentes

Tanto no SQL quanto no PySpark, podemos salvar os resultados em arquivos externos, como CSVs. No PySpark, usamos o método write.csv().

SQL

```sql
COPY (SELECT * FROM tabela) TO '/path/to/file.csv' CSV HEADER;
```

PySpark

```python
df.write.csv("/path/to/file.csv", header=True)
```

11 - Remoção de Colunas

Em PySpark, o método drop() é usado para remover colunas indesejadas de um DataFrame, de forma semelhante à exclusão implícita em SQL ao não selecionar as colunas.

SQL

```sql
SELECT coluna1, coluna2 FROM tabela;
```

PySpark

```python
df.drop("coluna3").show()
```

12 - Renomear Colunas

Renomear colunas em SQL é feito com AS. No PySpark, a função withColumnRenamed() é utilizada para dar um novo nome a uma coluna existente, facilitando a compreensão dos dados.

SQL

```sql
SELECT coluna1 AS nova_coluna1 FROM tabela;
```

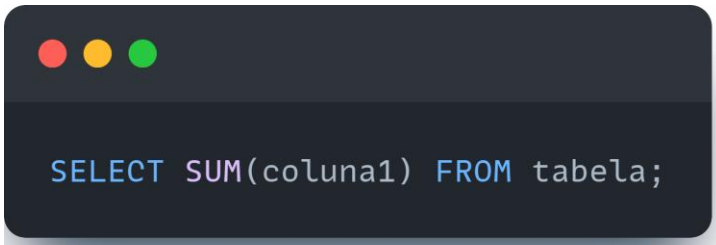
PySpark

```python
df.withColumnRenamed("coluna1", "nova_coluna1").show()
```

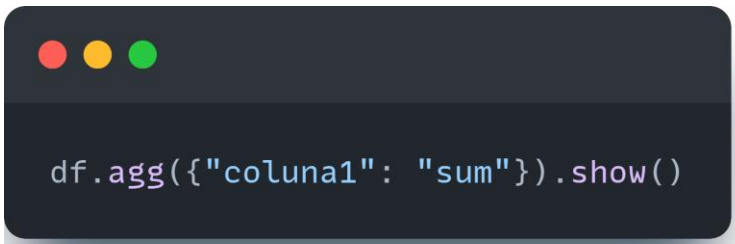
13 - Soma de Colunas

Tanto em SQL quanto em PySpark, somar os valores de uma coluna é uma prática comum para obter o total de um conjunto de dados. No PySpark, a função agg() permite aplicar uma operação de agregação, como sum, a uma coluna.

SQL



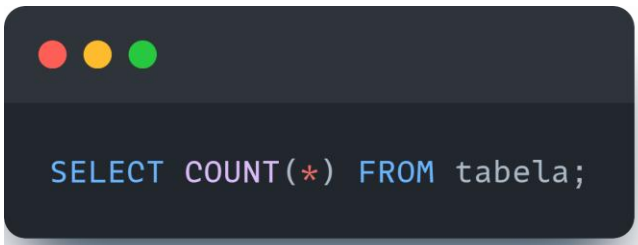
PySpark



14 - Contagem de Linhas

Contar o total de linhas em uma tabela ou DataFrame é uma operação básica de verificação de tamanho. Em PySpark, a função count() retorna o número de linhas presentes em um DataFrame.

SQL



PySpark

15 - Média de uma Coluna

Calcular a média dos valores em uma coluna é importante para análises estatísticas. Em PySpark, o método agg() é usado em conjunto com avg() para obter a média.

SQL

```sql
SELECT AVG(coluna1) FROM tabela;
```

PySpark

```python
df.agg({"coluna1": "avg"}).show()
```

16 - Mínimo e Máximo de uma Coluna

Identificar o menor e o maior valor em uma coluna é útil para conhecer os limites dos dados. No PySpark, isso é feito com agg() aplicando min() e max().

SQL

```sql
SELECT MIN(coluna1), MAX(coluna1) FROM tabela;
```

PySpark

```python
df.agg({"coluna1": "min", "coluna1": "max"}).show()
```

17 - Aplicar Função UDF (User Defined Function)

Funções definidas pelo usuário (UDFs) permitem aplicar transformações personalizadas aos dados. Em PySpark, as UDFs são criadas com udf() e podem ser aplicadas a colunas usando withColumn().

SQL

```sql
CREATE FUNCTION minha_funcao(...) RETURNS ...;
```

PySpark

```python
from pyspark.sql.functions import udf
from pyspark.sql.types import IntegerType

def minha_funcao(valor):
    return valor * 2

udf_minha_funcao = udf(minha_funcao, IntegerType())
df.withColumn("nova_coluna", udf_minha_funcao(df.coluna1)).show()
```

18 - Alterar Tipo de Coluna

Converter o tipo de uma coluna é uma operação comum para alinhar os dados com os tipos corretos. Em PySpark, cast() é usado em withColumn() para alterar o tipo de uma coluna específica.

SQL

```sql
ALTER TABLE tabela MODIFY coluna1 INT;
```

PySpark

```python
df.withColumn("coluna1", df["coluna1"].cast("int")).show()
```

19 - Concatenar Colunas

A concatenação de colunas permite combinar os valores de várias colunas em uma só. No PySpark, concat() é usado junto com lit() para adicionar espaços ou outros caracteres entre as colunas concatenadas.

SQL

```sql
SELECT CONCAT(coluna1, coluna2) AS nova_coluna FROM tabela;
```

PySpark

```python
from pyspark.sql.functions import concat, lit
df.withColumn("nova_coluna", concat(df.coluna1, lit(" "), df.coluna2)).show()
```

20 - Filtragem com Condições Múltiplas

A filtragem com múltiplas condições ajuda a refinar os dados com base em critérios específicos. Em PySpark, filter() aceita expressões lógicas combinadas com & (AND) e | (OR).

SQL

```sql
SELECT * FROM tabela WHERE coluna1 = 'valor' AND coluna2 > 10;
```

PySpark

```python
df.filter((df.coluna1 == 'valor') & (df.coluna2 > 10)).show()
```

21 - Filtro com Condições "OU"

Filtrar dados com condições de "OU" permite selecionar registros que atendam a pelo menos uma de várias condições. No PySpark, usamos | para definir essa lógica dentro do filter().

SQL

```sql
SELECT * FROM tabela WHERE coluna1 = 'valor1' OR coluna1 = 'valor2';
```

PySpark

```python
df.filter((df.coluna1 == 'valor1') | (df.coluna1 == 'valor2')).show()
```

22 - Criar Coluna com Condicional (IF ELSE)

Criar colunas com valores condicionais ajuda a categorizar dados com base em critérios específicos. No PySpark, when() permite aplicar lógicas IF-ELSE em uma nova coluna.

SQL

```sql
SELECT coluna1,
       CASE
           WHEN coluna1 > 10 THEN 'Alto'
           ELSE 'Baixo'
       END AS categoria
FROM tabela;
```

PySpark

```python
from pyspark.sql.functions import when

df.withColumn("categoria", when(df.coluna1 > 10, "Alto").otherwise("Baixo")).sho
```

23 - Agrupamento e Soma

O agrupamento de dados e a soma das colunas são operações importantes para sumarizar informações. No PySpark, groupBy() combinado com agg() permite realizar essas operações em colunas específicas.

SQL

```sql
SELECT coluna1, SUM(coluna2) FROM tabela GROUP BY coluna1;
```

PySpark

```python
df.groupBy("coluna1").agg({"coluna2": "sum"}).show()
```

24 - Contagem de Valores Únicos

Contar valores únicos em colunas ajuda a entender a diversidade de dados. No PySpark, countDistinct() é uma função de agregação útil para essa tarefa.

SQL

```sql
SELECT coluna1, COUNT(DISTINCT coluna2) FROM tabela GROUP BY coluna1;
```

PySpark

```python
df.groupBy("coluna1").agg({"coluna2": "countDistinct"}).show()
```

25 - Cruzamento de Dados (CROSS JOIN)

Um CROSS JOIN cria um produto cartesiano entre duas tabelas, combinando todas as linhas de ambas. Em PySpark, o método crossJoin() faz isso de forma eficiente.

SQL

```
df1.crossJoin(df2).show()
```

PySpark

```
SELECT * FROM tabela1 CROSS JOIN tabela2;
```

26 - Adicionar Coluna com Valor Constante

Adicionar uma coluna com um valor constante pode ser útil para criar uma referência comum em todas as linhas. Em PySpark, isso é feito com lit() em withColumn().

SQL

```sql
SELECT coluna1, 'valor_constante' AS nova_coluna FROM tabela;
```

PySpark

```python
from pyspark.sql.functions import lit

df.withColumn("nova_coluna", lit("valor_constante")).show()
```

27 - Contar Valores Nulos

Contar valores nulos ajuda a identificar problemas de qualidade dos dados. No PySpark, isNull() é usado dentro de filter() para contar quantas linhas têm valores nulos em uma coluna.

SQL

```sql
SELECT COUNT(*) FROM tabela WHERE coluna1 IS NULL;
```

PySpark

```
df.filter(df.coluna1.isNull()).count()
```

28 - Remover Valores Nulos

Remover valores nulos é uma prática comum para limpar os dados. Em PySpark, na.drop() pode ser usado para excluir linhas que contêm valores nulos em colunas específicas.

SQL

```sql
DELETE FROM tabela WHERE coluna1 IS NULL;
```

PySpark

```python
df.na.drop(subset=["coluna1"]).show()
```

29 - Preencher Valores Nulos

Substituir valores nulos por valores padrão ajuda a manter a consistência dos dados. Em PySpark, na.fill() é usado para preencher nulos em uma ou mais colunas com um valor específico.

SQL

```sql
df.na.fill("valor", subset=["coluna1"]).show()
```

PySpark

```sql
UPDATE tabela SET coluna1 = 'valor' WHERE coluna1 IS NULL;
```

30 - Filtragem com LIKE

A filtragem com LIKE permite buscar padrões específicos em strings. Em PySpark, o método like() ajuda a identificar registros que contêm uma substring específica.

SQL

```sql
SELECT * FROM tabela WHERE coluna1 LIKE '%valor%';
```

PySpark

```python
df.filter(df.coluna1.like("%valor%")).show()
```

31 - Substring de uma Coluna

Extrair partes de uma string é útil para trabalhar com segmentos de dados textuais. Em PySpark, a função substring() é usada para criar uma nova coluna com apenas uma parte da string original.

SQL

```sql
SELECT SUBSTRING(coluna1, 1, 5) FROM tabela;
```

PySpark

```python
from pyspark.sql.functions import substring

df.withColumn("nova_coluna", substring(df.coluna1, 1, 5)).show()
```

32 - Conversão de Data

Converter strings para objetos de data facilita o trabalho com operações de data e hora. No PySpark, to_date() é usado para transformar uma string em um objeto de data com base em um formato especificado.

SQL

```sql
SELECT TO_DATE(coluna_data, 'YYYY-MM-DD') FROM tabela;
```

PySpark

```python
from pyspark.sql.functions import to_date

df.withColumn("data_formatada", to_date(df.coluna_data, "yyyy-MM-dd")).show()
```

33 - Data e Hora Atuais

Adicionar a data e hora atual como uma coluna é útil para registrar quando um registro foi processado. Em PySpark, current_timestamp() cria uma nova coluna com a data e hora atual.

SQL

```sql
SELECT CURRENT_TIMESTAMP;
```

PySpark

```python
from pyspark.sql.functions import current_timestamp

df.withColumn("data_hora_atual", current_timestamp()).show()
```

34 - Extração de Mês ou Ano de uma Data

Extrair partes específicas de uma data, como mês ou ano, ajuda na análise temporal dos dados. Em PySpark, as funções month() e year() são usadas para criar novas colunas com essas informações.

SQL

```sql
SELECT EXTRACT(MONTH FROM coluna_data) FROM tabela;
```

PySpark

```python
from pyspark.sql.functions import month, year

df.withColumn("mes", month(df.coluna_data)).withColumn("ano", year(df.coluna_data)).show()
```

35 - Filtrar por Datas

Filtrar registros com base em datas permite análises com recortes temporais. No PySpark, filter() é usado para aplicar condições de comparação direta em colunas de data.

SQL

```sql
SELECT * FROM tabela WHERE coluna_data > '2023-01-01';
```

PySpark

```python
df.filter(df.coluna_data > '2023-01-01').show()
```

36 - Criar uma Tabela Temporária

Criar uma tabela temporária permite executar consultas SQL sobre um DataFrame. No PySpark, createOrReplaceTempView() cria uma tabela temporária que pode ser consultada com comandos SQL usando spark.sql().

SQL

```sql
CREATE TEMP TABLE tabela_temp AS SELECT * FROM tabela;
```

PySpark

```python
df.createOrReplaceTempView("tabela_temp")
```

37 - Executar SQL no PySpark

xecutar SQL diretamente no PySpark permite aplicar comandos SQL familiares em DataFrames. O método spark.sql() é usado para executar consultas em tabelas temporárias criadas anteriormente.

SQL

```sql
SELECT * FROM tabela;
```

PySpark

```python
spark.sql("SELECT * FROM tabela_temp").show()
```

38 - Remoção de Espaços em Branco

Remover espaços em branco de strings melhora a qualidade dos dados e evita problemas de comparação. Em PySpark, trim() remove espaços extras de uma coluna de texto.

SQL

```sql
SELECT TRIM(coluna1) FROM tabela;
```

PySpark

```python
from pyspark.sql.functions import trim

df.withColumn("coluna1_trim", trim(df.coluna1)).show()
```

39 - Agrupamento com Função HAVING

A cláusula HAVING em SQL é usada para filtrar os resultados de uma agregação. Em PySpark, a função filter() é aplicada após um groupBy() para fazer a filtragem.

SQL

```sql
SELECT coluna1, COUNT(*) FROM tabela GROUP BY coluna1 HAVING COUNT(*) > 10;
```

PySpark

```python
df.groupBy("coluna1").count().filter("count > 10").show()
```

40 - Operação com Janela (Window Function)

Funções de janela permitem cálculos avançados, como rankings e somas cumulativas, que são aplicados sobre um subconjunto de dados. Em PySpark, Window é combinado com funções como rank() para esses cálculos.

SQL

```sql
SELECT
coluna1,
RANK() OVER (PARTITION BY coluna2 ORDER BY coluna3 DESC) AS rank_col

FROM tabela;
```

PySpark

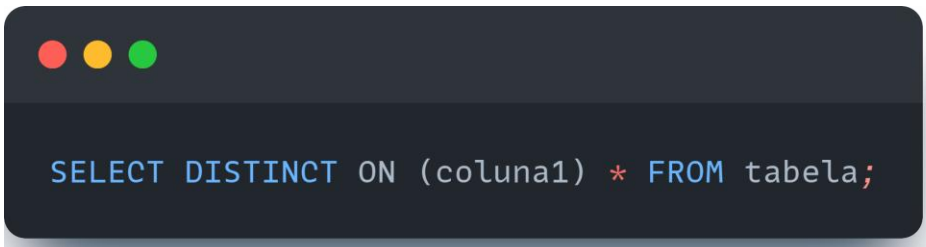
```python
from pyspark.sql.window import Window
from pyspark.sql.functions import rank

windowSpec = Window.partitionBy("coluna2").orderBy(df.coluna3.desc())
df.withColumn("rank_col", rank().over(windowSpec)).show()
```

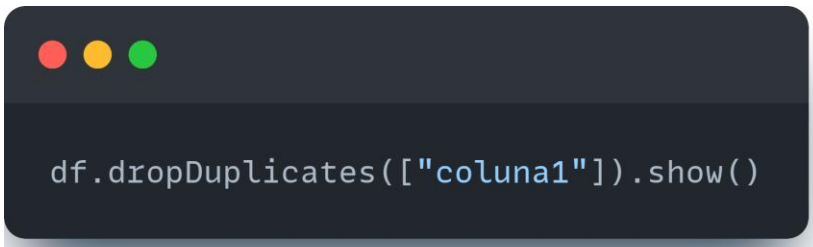
41 - Remover Duplicatas com Base em Colunas Específicas

Remover duplicatas com base em colunas específicas ajuda a garantir a unicidade dos dados. Em PySpark, dropDuplicates() remove linhas duplicadas com base nas colunas especificadas.

SQL



PySpark



42 - Concatenar DataFrames (UNION)

Concatenar DataFrames ou tabelas permite combinar conjuntos de dados com a mesma estrutura. Em PySpark, union() junta dois DataFrames, desde que tenham o mesmo esquema.

SQL

```sql
SELECT * FROM tabela1 UNION SELECT * FROM tabela2;
```

PySpark

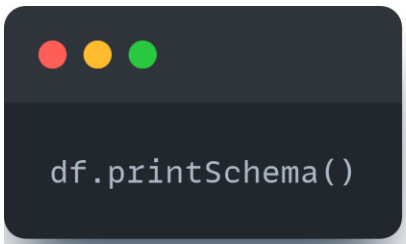
```python
df1.union(df2).show()
```

43 - Exibição de Esquema de DataFrame

Verificar o esquema de um DataFrame permite entender a estrutura dos dados, incluindo tipos de colunas. Em PySpark, printSchema() exibe uma representação hierárquica das colunas e seus tipos.

SQL

PySpark



44 - Mesclagem de Tabelas (MERGE)

O comando MERGE é usado para mesclar dados de duas tabelas com base em uma condição. Ele é útil para operações de upsert em que você deseja atualizar registros existentes e inserir novos. No PySpark, isso é feito com a biblioteca Delta Lake, que suporta operações merge.

SQL

```sql
MERGE INTO tabela_destino AS dest
USING tabela_fonte AS src
ON dest.id = src.id
WHEN MATCHED THEN
    UPDATE SET dest.coluna1 = src.coluna1
WHEN NOT MATCHED THEN
    INSERT (id, coluna1) VALUES (src.id, src.coluna1);
```

PySpark

```python
from delta.tables import DeltaTable

deltaTable = DeltaTable.forPath(spark, "/caminho/para/tabela_destino")
deltaTable.alias("dest").merge(
    df_fonte.alias("src"),
    "dest.id = src.id"
).whenMatchedUpdate(set={"coluna1": "src.coluna1"}) \
 .whenNotMatchedInsert(values={"id": "src.id", "coluna1": "src.coluna1"}) \
 .execute()
```

45 - Calcular Percentual de uma Coluna

Calcular percentuais ajuda a entender a representatividade dos valores de uma coluna em relação ao total. Em PySpark, é possível fazer isso criando uma coluna calculada usando operações matemáticas. 47. Aplicar Função de Janela Cumulativa SQL: sql Copiar código

SQL

```sql
SELECT (coluna1 / total) * 100 AS percentual FROM tabela;
```

PySpark

```python
total = df.agg({"coluna1": "sum"}).collect()[0][0]
df.withColumn("percentual", (df.coluna1 / total) * 100).show()
```

46 - Aplicar Função de Janela Cumulativa

Funções de janela cumulativas são usadas para somar valores sequencialmente em uma ordem específica. No PySpark, Window com sum().over() permite criar colunas com somas acumuladas.

SQL

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

windowSpec = Window.orderBy("coluna1")
df.withColumn("soma_acumulada", sum("coluna2").over(windowSpec)).show()
```

PySpark

```sql
SELECT coluna1, SUM(coluna2) OVER (ORDER BY coluna1) AS soma_acumulada
FROM tabela;
```

47 - Substituir Valores em uma Coluna

Substituir valores específicos em uma coluna é útil para limpar ou padronizar dados. Em PySpark, regexp_replace() troca partes de uma string com base em um padrão.

SQL

```sql
UPDATE tabela SET coluna1 = 'novo_valor' WHERE coluna1 = 'valor_antigo';
```

PySpark

```python
from pyspark.sql.functions import regexp_replace

df.withColumn("coluna1", regexp_replace(df.coluna1, "valor_antigo", "novo_valor")).show()
```

48 - Adicionar Índice a uma Tabela

Em SQL, criar um índice ajuda a melhorar o desempenho de consultas que usam colunas específicas. Em PySpark, embora não existam índices físicos como no SQL, você pode adicionar uma coluna de índice virtual com monotonically_increasing_id() para fins de identificação e ordenação.

SQL

```sql
CREATE INDEX idx_coluna1 ON tabela (coluna1);
```

PySpark

```python
from pyspark.sql.functions import monotonically_increasing_id

df_com_indice = df.withColumn("indice", monotonically_increasing_id())
df_com_indice.show()
```

49 - Agrupamento com ROLLUP

O ROLLUP é uma extensão do GROUP BY que permite gerar subtotais em diferentes níveis de agregação. Em PySpark, isso é implementado com a função rollup(), que cria um agrupamento hierárquico com colunas especificadas, permitindo cálculos de somas parciais em vários níveis.

SQL

```sql
SELECT coluna1, coluna2, SUM(coluna3)
FROM tabela
GROUP BY ROLLUP (coluna1, coluna2);
```

PySpark

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

df_grouped = df.groupBy("coluna1", "coluna2") \
    .agg(sum("coluna3").alias("soma_coluna3"))

df_rolled_up = df_grouped.rollup("coluna1", "coluna2") \
    .sum()

df_rolled_up.show()
```

50 - Pivotar Tabelas (Pivot Table)

A operação de pivot transforma linhas em colunas. Em SQL, a função PIVOT é usada para criar colunas dinâmicas com base nos valores de uma coluna específica e aplicar uma agregação. Em PySpark, o método pivot() combina a função groupBy() com a agregação desejada (neste caso, sum()).

SQL

```sql
SELECT coluna1,
       SUM(CASE WHEN coluna2 = 'valor1' THEN coluna3 ELSE 0 END) AS valor1,
       SUM(CASE WHEN coluna2 = 'valor2' THEN coluna3 ELSE 0 END) AS valor2
FROM tabela
GROUP BY coluna1;
```

PySpark

```python
df.groupBy("coluna1").pivot("coluna2").sum("coluna3").show()
```

Quer fazer o download dos códigos?

Baixe os Códigos Agora!

Os exemplos apresentados neste e-book são apenas o começo. Para facilitar ainda mais sua jornada, todos os 50 códigos SQL transformados para PySpark estão disponíveis para download.

Clique no link abaixo para baixar os códigos e começar a usar diretamente em seus projetos:

BAIXAR OS CÓDIGOS

Com os códigos em mãos, você pode testar, modificar e aplicar em seu próprio ambiente de dados com muito mais agilidade. Não perca tempo digitando, aproveite esses exemplos prontos!

Esses 50 comandos PySpark, junto com as explicações e exemplos, oferecem uma base sólida e cobrem uma ampla gama de operações de manipulação e análise de dados no Microsoft Fabric.

Este eBook apresentou uma comparação prática entre SQL e PySpark para diversos cenários de manipulação de dados, focando em sua aplicação dentro do Microsoft Fabric. Com esta comparação, profissionais de dados podem transitar entre as duas linguagens com mais facilidade e aproveitar ao máximo o poder do PySpark dentro de um ambiente moderno de análise de dados.

Para quem já é familiarizado com SQL, aprender PySpark é um passo importante para lidar com grandes volumes de dados de maneira mais eficiente e escalável. O Microsoft Fabric oferece uma plataforma flexível que permite aproveitar ambas as abordagens.


DataFabric
datafabric.com.br