



Salve para mais tarde



Compartilhe
conhecimento

Operações básicas

PANDAS E PYSPARK



Oi,
Marcelo!



CONTEÚDO

1. <u>Read Files</u>	03
2. <u>Description</u>	04
3. <u>Select columns</u>	07
4. <u>Add Columns</u>	09
5. <u>Rename columns</u>	11
6. <u>Order</u>	13
7. <u>Filter</u>	15
8. <u>JOIN DataFrame</u>	17
9. <u>Concatenate DataFrame</u>	19
10. <u>Missing Values</u>	22
11. <u>Deduplicate</u>	24
12. <u>Data type conversion</u>	25
13. <u>GroupBy e Aggregation agg</u>	27
14. <u>Pivot Table</u>	30
15. <u>Replace values</u>	32
16. <u>Functions</u>	34
17. <u>Regex</u>	36
18. <u>Window Function</u>	40
19. <u>Export Data</u>	42
20. <u>Referencias</u>	44

1. Read files

PANDAS

```
import pandas as pd
df_pandas = pd.read_csv("arquivo.csv")
df_pandas = pd.read_parquet("arquivo.parquet")
df_pandas = pd.read_json("arquivo.json")
```

PYSPARK

```
from pyspark.sql import SparkSession

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

df_spark = spark.read.option("header", True).csv("arquivo.csv")
df_json = spark.read.json("arquivo.json")
df_parquet = spark.read.parquet("arquivo.parquet")
```

2. Description

PANDAS

```
# Number_row
num_row = len(df) # Conta o número de elementos no índice do DataFrame

# Statistic
df.describe() # Apenas colunas numéricas

df.describe(include="all") # Retorna colunas categóricas (strings)

# Info
df.info() # Retorna colunas, tipos de dados, valores não nulos e uso de memória
```

Diferença entre `len(df)` e `df.shape[0]`

Ambos retornam o número de linhas, mas há uma pequena diferença interna:

- ✓ `len(df)`: Conta o número de elementos no índice do DataFrame.
- ✓ `df.shape[0]`: Retorna diretamente o número de linhas como parte da

estrutura do DataFrame.

Qual usar?

Na prática, ambos funcionam da mesma forma para contar linhas em um DataFrame normal, mas `shape[0]` pode ser ligeiramente mais rápido porque **acessa diretamente a estrutura do objeto**, enquanto `len(df)` precisa **contar os elementos no índice**.

PYSPARK

```
# Row numbers:  
num_row = df_spark.count() # Conta o número de linhas  
print(f"Número de linhas: {num_row}")
```

✓ No PySpark, usamos `.count()`, pois `len(df_spark)` não funciona.

```
# Statistic  
df_spark.describe().show() # Apenas colunas numéricas
```

⚠ Diferente do Pandas, o `describe()` do PySpark só retorna estatísticas de colunas numéricas.

Se precisar incluir colunas categóricas, você pode contar valores únicos:

```
from pyspark.sql.functions import countDistinct  
df_spark.select([countDistinct(c).alias(c) for c in df_spark.columns]).show()
```

No PySpark, o `.select()` aceita uma lista de colunas ou expressões para seleção. Como queremos aplicar `countDistinct()` a cada coluna do DataFrame, usamos **list comprehension** para gerar essa lista dinamicamente.

1 `df_spark.columns` → Retorna uma lista com os nomes das colunas.

◆ Exemplo: `['idade', 'salario', 'cidade']`

2 `countDistinct(c).alias(c)` → Para cada coluna `c`, aplicamos `countDistinct()`, que conta valores únicos na coluna e renomeia o resultado com `.alias(c)`.

◆ Exemplo: `countDistinct("idade").alias("idade")`

3 `[countDistinct(c).alias(c) for c in df_spark.columns]` → Cria uma lista de expressões para cada coluna.

[O que é uma list comprehension?](#)

```
df_spark.printSchema() # Mostra colunas, tipos de dados e estrutura
```

⚠ No PySpark, não existe `df.info()` como no Pandas.

O mais próximo disso é:

- **`df_spark.printSchema()`**: Exibe os tipos das colunas.
- **`df_spark.describe().show()`**: Estatísticas básicas.
- **`df_spark.select([count(c).alias(c) for c in df_spark.columns]).show()`**: Conta valores não nulos.

📌 Resumo Comparativo

Função	Pandas	PySpark
Número de linhas	<code>len(df)</code> ou <code>df.shape[0]</code>	<code>df_spark.count()</code>
Estatísticas	<code>df.describe()</code>	<code>df_spark.describe().show()</code> (apenas numéricas)
Info	<code>df.info()</code>	<code>df_spark.printSchema()</code>

3. Select columns

PANDAS

```
# Seleciona uma única coluna
df["coluna1"]

# Se o nome da coluna não tiver espaços ou caracteres especiais
df.coluna1

# Múltiplas colunas
df[["coluna1", "coluna2"]]

# Seleciona coluna por índice
df.iloc[:, [0, 2]] # Seleciona a primeira e a terceira coluna

# Seleciona coluna com filtro
df.select_dtypes(include="number") # No exemplo é coluna numérica
```

PYSPARK

Em PySpark, a seleção de colunas é feita com `.select()` ou `.drop()`

```
# Seleciona uma única coluna
df_spark.select("coluna1").show()

# Seleciona múltiplas colunas
df_spark.select("coluna1", "coluna2").show()

# Seleciona múltiplas columns com lista
colunas = ["coluna1", "coluna2"]
df_spark.select(colunas).show()
```

```
# Seleciona coluna com expressão SQL
from pyspark.sql.functions import col

df_spark.select(col("coluna1"), col("coluna2")).show()

# Seleciona todas as colunas exceto (Drop)
df_spark.drop("coluna3").show()

# Seleciona colunas dinamicamente
# No exemplo apenas colunas numéricas:
numeric_cols = [c for c, t in df_spark.dtypes if t in ["int", "double", "float"]]
df_spark.select(numeric_cols).show()
```

Resumo Comparativo

Ação	Pandas	PySpark
Selecionar 1 coluna	<code>df["coluna1"]</code>	<code>df_spark.select("coluna1")</code>
Selecionar múltiplas colunas	<code>df[["coluna1", "coluna2"]]</code>	<code>df_spark.select("coluna1", "coluna2")</code>
Selecionar por índice	<code>df.iloc[:, [0, 2]]</code>	✗ (PySpark não usa índices)
Selecionar por tipo	<code>df.select_dtypes(include="number")</code>	<code>df_spark.select([c for c, t in df_spark.dtypes if t in ["int", "double", "float"]])</code>
Excluir colunas	<code>df.drop("coluna3", axis=1)</code>	<code>df_spark.drop("coluna3")</code>

4. Add columns

PANDAS

```
# Adiciona uma coluna nova com valor fixo
import pandas as pd

df = pd.DataFrame({"A": [1, 2, 3]})
df["nova_coluna"] = 100 # Adiciona uma nova coluna com valor 100
print(df)

# Adicionar uma coluna com cálculo baseado em outra
df["dobro_A"] = df["A"] * 2

# Adicionar uma coluna com valores condicionais
df["categoria"] = df["A"].apply(lambda x: "Alto" if x > 2 else "Baixo")
```

PYSPARK

```
# Adicionar coluna
from pyspark.sql import SparkSession
from pyspark.sql.functions import col

df = df.withColumn("Nova_coluna", col("coluna_antiga") * 2) # A nova coluna vai ter o dobro de valores da coluna antiga

# Adicionar uma coluna com valor fixo
from pyspark.sql import SparkSession
from pyspark.sql.functions import lit

spark = SparkSession.builder.getOrCreate()

df_spark = spark.createDataFrame([(1,), (2,), (3,)], ["A"])
df_spark = df_spark.withColumn("nova_coluna", lit(100))
df_spark.show()

# Adicionar uma coluna com cálculo
from pyspark.sql.functions import col

df_spark = df_spark.withColumn("dobro_A", col("A") * 2)
df_spark.show()

# Adicionar uma coluna com valores condicionais (when)
from pyspark.sql.functions import when

df_spark = df_spark.withColumn("categoria", when(col("A") > 2, "Alto").otherwise("Baixo"))
df_spark.show()
```

col("coluna_antiga")

- Representa a coluna "coluna_antiga" no DataFrame PySpark.
- Diferente do Pandas, no PySpark não usamos diretamente df["coluna"], pois ele trabalha com expressões SQL.

```
# Criando um DF em pyspark e adicionando nova coluna
from pyspark.sql import SparkSession
from pyspark.sql.functions import col

# Criando sessão Spark
spark = SparkSession.builder.getOrCreate()

# Criando DataFrame
data = [(1,), (2,), (3,)]
df = spark.createDataFrame(data, ["coluna_antiga"])

# Adicionando a nova coluna
df = df.withColumn("Nova_coluna", col("coluna_antiga") * 2)

# Mostrando resultado
df.show()
```

📌 Resumo

Ação	Pandas	PySpark
Adicionar coluna fixa	<code>df["nova"] = 100</code>	<code>df = df.withColumn("nova", lit(100))</code>
Adicionar coluna com cálculo	<code>df["dobro"] = df["A"] * 2</code>	<code>df = df.withColumn("dobro", col("A") * 2)</code>
Adicionar coluna condicional	<code>df["cat"] = df["A"].apply(lambda x: "Alto" if x > 2 else "Baixo")</code>	<code>df = df.withColumn("cat", when(col("A") > 2, "Alto").otherwise("Baixo"))</code>
Adicionar uma cópia de uma coluna	<code>df["Nova_coluna"] = df["coluna_antiga"]</code>	<code>df = df.withColumn("Nova_coluna", col("coluna_antiga"))</code>

5. Rename Columns

PANDAS

No pandas, usamos o método `.rename()`.

```
# Renomear uma coluna
df = df.rename(columns={"coluna_antiga": "coluna_nova"})

# Renomear todas as colunas de uma vez
df.columns = ["nova_coluna"] # Apenas se quiser substituir todas as colunas

# Renomear múltiplas colunas
df = df.rename(columns={"coluna1": "nova1", "coluna2": "nova2"})
```

PYSPARK

No PySpark, usamos `.withColumnRenamed()`.

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.getOrCreate()

df_spark = df_spark.withColumnRenamed("coluna_antiga", "coluna_nova")

# Renomear múltiplas colunas de uma vez
df_spark = df_spark.selectExpr("coluna_antiga as coluna_nova")

# Renomear várias colunas dinamicamente
df_spark = df_spark.selectExpr(["coluna1 as nova1", "coluna2 as nova2"])
```

Ação	Pandas	PySpark
Renomear uma coluna	<pre>df.rename(columns={ "coluna_antiga": "coluna_nova"})</pre>	<pre>df_spark.withColumnR enamed("coluna_antig a", "coluna_nova")</pre>
Renomear várias colunas	<pre>df.rename(columns={ "c1": "n1", "c2": "n2"})</pre>	<pre>df_spark.selectExpr("c1 as n1", "c2 as n2")</pre>

6. Order

PANDAS

Em pandas, podemos reorganizar as colunas usando **df[lista_de_colunas]**.

```
# Ordenar colunas em ordem alfabética
df = df[sorted(df.columns)]

# Reordenar colunas manualmente
df = df[["C", "A", "B"]] # Define a ordem desejada
```

PYSPARK

No PySpark, usamos **.select()** e fornecemos a nova ordem das colunas.

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.getOrCreate()

df_spark = df_spark.select(sorted(df_spark.columns))

# Reordenar colunas manualmente
df_spark = df_spark.select("C", "A", "B") # Define a ordem desejada
```

```
# Ordenar valores em ordem decrescente
from pyspark.sql import SparkSession
from pyspark.sql.functions import desc

df_spark.orderBy(desc("valor")).show()
```

O desc no PySpark é usado para ordenar colunas em ordem decrescente quando utilizamos **.orderBy()**. Ele vem do módulo `pyspark.sql.functions`

Resumo

Ação	Pandas	PySpark
Ordenar colunas em ordem alfabética	<pre>df = df[sorted(df.columns)]</pre>	<pre>df_spark = df_spark.select(sorted(df_spark.columns))</pre>
Reordenar colunas manualmente	<pre>df = df[["C", "A", "B"]]</pre>	<pre>df_spark = df_spark.select("C", "A", "B")</pre>

Resumo Ordenar com Pyspark

Ação	PySpark
Ordenar crescente (padrão)	<pre>df_spark.orderBy("coluna")</pre>
Ordenar decrescente	<pre>df_spark.orderBy(desc("coluna"))</pre>

7. Filter

PANDAS

Usamos `df[condicao]` para filtrar linhas.

```
# Filtrar valores maiores que 10
import pandas as pd

df = pd.DataFrame({"A": [5, 15, 20], "B": [2, 25, 30]})

# Filtrar linhas onde A > 10
df_filtrado = df[df["A"] > 10]

print(df_filtrado)

# Múltiplas condições (AND)
df[(df["A"] > 10) & (df["B"] < 30)]

# Múltiplas condições (OR)
df[(df["A"] > 10) | (df["B"] < 30)]

# Filtrar valores específicos
df[df["A"].isin([15, 20])]
```

PYSPARK

```
# Filtrar valores maiores que 10

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

spark = SparkSession.builder.getOrCreate()

df_spark = spark.createDataFrame([(5, 2), (15, 25), (20, 30)], ["A", "B"])

# Filtrar onde A > 10
df_filtrado = df_spark.filter(col("A") > 10)

df_filtrado.show()

# Múltiplas condições (AND)
df_spark.filter((col("A") > 10) & (col("B") < 30))

# Múltiplas condições (OR)
df_spark.filter((col("A") > 10) | (col("B") < 30))

# Filtrar valores específicos
df_spark.filter(col("A").isin([15, 20]))
```

Ação	Pandas	PySpark
Filtrar valores > 10	<code>df[df["A"] > 10]</code>	<code>df_spark.filter(col("A") > 10)</code>
Filtrar com AND	<code>df[(df["A"] > 10) & (df["B"] < 30)]</code>	<code>df_spark.filter((col("A") > 10) & (col("B") < 30))</code>
Filtrar com OR	<code>`df[(df["A"] > 10)</code>	<code>(df["B"] < 30)]`</code>
Filtrar valores específicos	<code>df[df["A"].isin([15, 20])]</code>	<code>df_spark.filter(col("A").isin([15, 20]))</code>

8. JOIN DataFrame

PANDAS

Usamos `pd.merge(df1, df2, on="coluna", how="tipo_do_join")`.

```
import pandas as pd

df1 = pd.DataFrame({"id": [1, 2, 3], "valor": ["A", "B", "C"]})
df2 = pd.DataFrame({"id": [2, 3, 4], "descricao": ["X", "Y", "Z"]})

# Inner Join (apenas correspondências)
df_inner = pd.merge(df1, df2, on="id", how="inner")

print(df_inner)

# Left Join
how="left"

# Right Join
how="right"

# Full Outer Join
how="outer"
```

PYSPARK

Usamos `join(df2, on="coluna", how="tipo_do_join")`.

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.getOrCreate()

df1_spark = spark.createDataFrame([(1, "A"), (2, "B"), (3, "C")], ["id", "valor"])
df2_spark = spark.createDataFrame([(2, "X"), (3, "Y"), (4, "Z")], ["id", "descricao"])

# Inner Join (apenas correspondências)
df_inner_spark = df1_spark.join(df2_spark, on="id", how="inner")

df_inner_spark.show()

# Left Join
how="left"

# Right Join
how="right"

# Full Outer Join
how="outer"

# Cross Join
how="cross"
```

Tipo de Join	Pandas	PySpark
Inner Join	<code>pd.merge(df1, df2, on="id", how="inner")</code>	<code>df1_spark.join(df2_spark, on="id", how="inner")</code>
Left Join	<code>pd.merge(df1, df2, on="id", how="left")</code>	<code>df1_spark.join(df2_spark, on="id", how="left")</code>
Right Join	<code>pd.merge(df1, df2, on="id", how="right")</code>	<code>df1_spark.join(df2_spark, on="id", how="right")</code>
Outer Join	<code>pd.merge(df1, df2, on="id", how="outer")</code>	<code>df1_spark.join(df2_spark, on="id", how="outer")</code>

9. Concatenate DataFrames

PANDAS

Usamos `pd.concat()` para unir DataFrames verticalmente ou horizontalmente.

```
1 # Concatenar DataFrames verticalmente (empilhar linhas)
2
3 # Concatenar (por padrão, axis=0 → empilha linhas)
4 df_concat = pd.concat([df1, df2])
5
6
7 import pandas as pd
8
9 df1 = pd.DataFrame({"id": [1, 2], "valor": ["A", "B"]})
10 df2 = pd.DataFrame({"id": [3, 4], "valor": ["C", "D"]})
11
12 # Concatenar (por padrão, axis=0 → empilha linhas)
13 df_concat = pd.concat([df1, df2])
14
15 print(df_concat)
16
17 # Resetar o índice após a concatenação
18 pd.concat([df1, df2], ignore_index=True)
19
20 # Concatenar horizontalmente (mesclar colunas)
21 pd.concat([df1, df2], axis=1)
```

O `pd.concat([df1, df2], axis=1)` simplesmente coloca os DataFrames lado a lado, sem criar combinações extras de linhas.

PYSPARK

Usamos **.union()** ou **.unionByName()** para empilhar DataFrames (apenas se tiverem os mesmos esquemas).

```
# Concatenar Verticalmente (Empilhar Linhas)
novo_df = df1.union(df2)
```

📌 O que faz?

- Junta os DataFrames empilhando as linhas (como `pd.concat([df1, df2])` em Pandas).
- Requisitos: Os DataFrames devem ter o mesmo esquema (mesmo número e tipo de colunas).

⚠ Se quiser remover duplicatas, use `unionByName()` e `distinct()`

```
400
401 novo_df = df1.unionByName(df2).distinct()
402
```

📌 Se os DataFrames têm uma chave comum, usamos **join()**

```
# Concatenar Horizontalmente (Empilhar Colunas)
novo_df = df1.join(df2, on="id", how="inner") # Faz um INNER JOIN pela coluna "id"
```

Isso não concatena colunas arbitrariamente, apenas junta os DataFrames baseado na chave "id"

Se NÃO houver uma chave comum e quiser apenas empilhar colunas

```
# Juntar colunas sem chave
from pyspark.sql.functions import monotonically_increasing_id

df1 = df1.withColumn("index", monotonically_increasing_id())
df2 = df2.withColumn("index", monotonically_increasing_id())

novo_df = df1.join(df2, on="index", how="inner").drop("index")
```

No exemplo, como PySpark não tem índices como Pandas, precisamos criar uma coluna de índice temporária para garantir que as linhas se alinhem corretamente.

Resumo

Ação	Pandas	PySpark (Correção)
Concatenar Linhas (Verticalmente)	<code>pd.concat([df1, df2])</code>	<code>df1.union(df2)</code>
Concatenar Colunas (Horizontalmente) Sem Chave	<code>pd.concat([df1, df2], axis=1)</code>	Criar índice e usar <code>join()</code>
Concatenar Colunas (Horizontalmente) Com Chave	<code>pd.merge(df1, df2, on="id")</code>	<code>df1.join(df2, on="id")</code>

Resumo

Ação	Descrição	Exemplo Pandas
Concatenar Verticalmente	Adiciona linhas, sem combinar colunas	<code>pd.concat([df1, df2])</code>
Concatenar Horizontalmente	Junta DataFrames lado a lado, sem levar em conta colunas em comum	<code>pd.concat([df1, df2], axis=1)</code>
Mesclar (Merge/Join)	Une DataFrames baseando-se em colunas em comum, como um JOIN no SQL	<code>pd.merge(df1, df2, on="id", how="inner")</code>

10. Missing Values

10.1 Identificar Valores Ausentes

PANDAS

```
# Conta valores ausentes por coluna
df.isnull().sum()

# Filtra linhas com pelo menos um valor ausente
df[df.isnull().any(axis=1)]
```

PYSPARK

```
5
6 from pyspark.sql.functions import col
7
8 # Mostra True/False para valores ausentes
9 df_spark.select([col(c).isNull().alias(c) for c in df_spark.columns]).show()
10
11 # Filtra linhas onde a coluna tem valores ausentes
12 df_spark.filter(df_spark["coluna"].isNull()).show()
13
```

10.2 Remover Valores Ausentes

PANDAS

```
# Remove linhas com valores ausentes
df.dropna()

# Remove valores ausentes apenas de uma coluna específica
df.dropna(subset=["coluna"])

# Remove colunas inteiras se tiverem valores ausentes
df.dropna(axis=1)
```

PYSPARK

```
# Remove linhas com valores ausentes
df_spark.na.drop()

# Remove valores ausentes apenas de uma coluna específica
df_spark.na.drop(subset=["coluna"])
```

10.3 Preencher Valores Ausentes

PANDAS

```
# Substitui valores ausentes por 0
df.fillna(0)

# Preenche valores ausentes apenas de uma coluna específica
df.fillna({"coluna": "valor"})

# Preenche com o valor anterior (forward fill)
df.ffill()

# Preenche com o próximo valor (backward fill)
df.bfill()
```

PYSPARK

```
# Substitui valores ausentes por 0
df_spark.na.fill(0)

# Preenche valores ausentes apenas de uma coluna específica
df_spark.na.fill({"coluna": "valor"})
```

11. Remove duplicates (Deduplicate)

PANDAS

```
# Remove linhas duplicadas considerando todas as colunas
df.drop_duplicates()

# Remove duplicatas considerando apenas colunas específicas
df.drop_duplicates(subset=["coluna1", "coluna2"])

# Mantém a primeira ocorrência (padrão)
df.drop_duplicates(keep="first")

# Mantém a última ocorrência
df.drop_duplicates(keep="last")

# Remove todas as duplicatas, sem manter nenhuma
df.drop_duplicates(keep=False)
```

```
# Remove duplicatas com base em colunas específicas e modifica o DataFrame original
df.drop_duplicates(subset=["coluna1", "coluna2"], inplace = True)
```

subset=["coluna1", "coluna2"]: Considera apenas essas colunas para detectar duplicatas.

inplace=True: Modifica o próprio df, sem precisar atribuir a uma nova variável.

⚠ Se **inplace=False**, você precisa salvar o resultado em uma nova variável

PYSPARK

```
# Remove linhas duplicadas considerando todas as colunas
df_spark.dropDuplicates()

# Remove duplicatas considerando apenas colunas específicas
df_spark.dropDuplicates(["coluna1", "coluna2"])
```


12. Data type conversion / Type Casting

PANDAS

No Pandas, usamos **.astype()** para converter tipos de coluna

```
import pandas as pd

# Criando um DataFrame
df = pd.DataFrame({"coluna": ["1", "2", "3"]})

# Convertendo para inteiro
df["coluna"] = df["coluna"].astype(int)

# Convertendo para float
df["coluna"] = df["coluna"].astype(float)

# Convertendo para string
df["coluna"] = df["coluna"].astype(str)
```

PYSPARK

No PySpark, usamos **.cast()** para fazer a conversão de tipos

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

# Criando a sessão Spark
spark = SparkSession.builder.appName("Exemplo").getOrCreate()

# Criando um DataFrame
data = [("1",), ("2",), ("3",)]
df_spark = spark.createDataFrame(data, ["coluna"])

# Convertendo para inteiro
df_spark = df_spark.withColumn("coluna", col("coluna").cast("int"))

# Convertendo para float
df_spark = df_spark.withColumn("coluna", col("coluna").cast("float"))

# Convertendo para string
df_spark = df_spark.withColumn("coluna", col("coluna").cast("string"))

df_spark.show()
```

Diferença Entre Pandas e PySpark

Ação	Pandas (astype())	PySpark (cast())
Converter para int	<code>df["coluna"].astype(int)</code>	<code>df_spark.withColumn("coluna", col("coluna").cast("int"))</code>
Converter para float	<code>df["coluna"].astype(float)</code>	<code>df_spark.withColumn("coluna", col("coluna").cast("float"))</code>
Converter para str	<code>df["coluna"].astype(str)</code>	<code>df_spark.withColumn("coluna", col("coluna").cast("string"))</code>

13. GroupBy e Aggregation agg

PANDAS

No Pandas, usamos **.groupby()** seguido de **.agg()** ou funções como **.sum()**, **.mean()**, etc

```
df.groupby('col').agg({'col': 'sum'})

# Exemplo
df_grouped = df.groupby("categoria").agg({"valor": "sum"})
```

A função **.agg()** recebe um dicionário, onde:

- A chave é a coluna a ser agregada.
- O valor é a função de agregação a ser aplicada (ex: 'sum', 'mean', 'max', etc.).

```
df_grouped = df.groupby("categoria")["valor"].sum()

# Usando .agg() para múltiplas agregações
df_grouped = df.groupby("categoria").agg({"valor": ["sum", "mean", "max"]})
print(df_grouped)
```

PSPARK

No PySpark, usamos **.groupBy()** seguido de **.agg()**, passando funções do módulo **pyspark.sql.functions**

```
# Exemplo com soma
from pyspark.sql.functions import sum

df_grouped = df_spark.groupBy('col').agg(sum('col2'))
df_grouped.show()
```

- **groupBy('col')**: Agrupa os dados pela coluna col.
- **agg(sum('col2'))**: Aplica a função **sum()** na coluna col2.

```

35
36 # Exemplo com várias funções
37 from pyspark.sql.functions import sum, avg, max
38
39 df_grouped = df_spark.groupBy('col').agg(
40     sum('col2').alias('soma'),
41     avg('col2').alias('media'),
42     max('col2').alias('maximo')
43 )
44
45 df_grouped.show()

```

Exemplo:

```

# Usando .agg() com funções agregadas

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

# Criando a sessão Spark
spark = SparkSession.builder.appName("Exemplo").getOrCreate()

# Criando um DataFrame Spark
data = [("A", 10), ("B", 20), ("A", 30), ("B", 40), ("A", 50)]
df_spark = spark.createDataFrame(data, ["categoria", "valor"])

# Agrupar por "categoria" e calcular soma, média e valor máximo
df_grouped = df_spark.groupBy("categoria").agg(
    sum("valor").alias("soma"),
    avg("valor").alias("media"),
    max("valor").alias("maximo")
)

df_grouped.show()

```

Diferença Entre Pandas e PySpark

Função	Pandas (groupby())	PySpark (groupBy())
Agrupar e somar	<pre>df.groupby("categoria") ["valor"].sum()</pre>	<pre>df_spark.groupBy("categoria").agg(sum("valor"))</pre>
Agrupar e múltiplas funções	<pre>df.groupby("categoria"). agg({"valor": ["sum", "mean", "max"]})</pre>	<pre>df_spark.groupBy("categoria").agg(sum("valor"), avg("valor"), max("valor"))</pre>

14. Pivot Table

A pivot table transforma linhas em colunas com base em uma agregação

PANDAS

```
0  # Criando uma pivot table
1  df_pivot = df.pivot_table(
2      index="categoria",
3      columns="ano",
4      values="valor",
5      aggfunc="sum")
6
7  print(df_pivot)
8
```

- **index="categoria"** : Mantém as categorias como índice.
- **columns="ano"** : Transforma valores únicos de ano em colunas.
- **values="valor"** : Dados que serão agregados.
- **aggfunc="sum"** Se houver múltiplos valores, ele soma.

PYSPARK

```
51
52 from pyspark.sql import SparkSession
53 from pyspark.sql.functions import sum
54
55 # Criando sessão Spark
56 spark = SparkSession.builder.appName("PivotExample").getOrCreate()
57
58 # Criando DataFrame
59 data = [
60     ("A", 2020, 10),
61     ("A", 2021, 20),
62     ("B", 2020, 30),
63     ("B", 2021, 40),
64     ("C", 2021, 50)
65 ]
66
67 df_spark = spark.createDataFrame(data, ["categoria", "ano", "valor"])
68
69 # Aplicando pivot
70 df_pivot_spark = df_spark.groupBy("categoria").pivot("ano").agg(sum("valor"))
71
72 df_pivot_spark.show()
73
```

- **groupBy("categoria")** : Mantém categoria como índice.
- **.pivot("ano")** : Transforma ano em colunas.
- **.agg(sum("valor"))**: Soma os valores de valor para cada combinação.

15. Replace values

O método `.replace()` permite substituir um ou mais valores em um `DataFrame` ou `Series`.

PANDAS

```
import pandas as pd

df['col'].replace(['valor_antigo1', 'valor_antigo2'], 'novo_valor', inplace = True)
# O argumento inplace=True faz a alteração diretamente no df (sem precisar atribuir a uma nova variável).
```

Exemplo:

```
09
10 import pandas as pd
11
12 # Criando um DataFrame de exemplo
13 data = {'col': ['A', 'B', 'C', 'D', 'A', 'C']}
14 df = pd.DataFrame(data)
15
16 # Substituindo múltiplos valores na coluna "col"
17 df['col'].replace(['A', 'C'], 'Z', inplace=True)
18
19 print(df)
20
21
```

`df['col'].replace(['A', 'C'], 'Z', inplace=True)`

- Substitui "A" e "C" por "Z" na coluna "col".
- O argumento **`inplace=True`** faz a alteração diretamente no `df` (sem precisar atribuir a uma nova variável).

PYSPARK

Para substituir Valores em PySpark (**replace** e **when**)

```
✓ from pyspark.sql import SparkSession
  from pyspark.sql.functions import when

# Criando sessão Spark
spark = SparkSession.builder.appName("ReplaceExample").getOrCreate()

# Criando DataFrame
data = [("A", 10), ("B", 20), ("C", 30), ("D", 40)]
df_spark = spark.createDataFrame(data, ["col1", "col2"])

# Substituir um valor específico
df_spark = df_spark.replace("A", "Z", subset=["col1"])

# Substituir múltiplos valores com replace
df_spark = df_spark.replace({"B": "Y", "C": "X"}, subset=["col1"])

# Substituir com when (útil para condições complexas)
df_spark = df_spark.withColumn("col2", when(df_spark["col2"] == 10, 100).otherwise(df_spark["col2"]))

df_spark.show()
```

- **.replace("A", "Z", subset=["col1"])** : Troca "A" por "Z".
- **.replace({"B": "Y", "C": "X"}, subset=["col1"])** : Substitui múltiplos valores.
- **when(df_spark["col2"] == 10, 100).otherwise(df_spark["col2"])**: Substitui 10 por 100 na coluna col2.

16. Funtions

PANDAS

Função em Pandas (apply com lambda)

```
df['nova_coluna'] = df['coluna_antiga'].apply(lambda x: x * 2)
```

```
import pandas as pd

# Criando DataFrame de exemplo
data = {'coluna_antiga': [1, 2, 3, 4, 5]}
df = pd.DataFrame(data)

# Aplicando uma função lambda para dobrar os valores
df['nova_coluna'] = df['coluna_antiga'].apply(lambda x: x * 2)

print(df)
```

.apply(lambda x: x * 2): Multiplica cada valor da coluna coluna_antiga por 2 e cria nova_coluna

PYSPARK

Função em PySpark com UDF (User Defined Function)

```
# Criando uma UDF de Integer

from pyspark.sql import SparkSession
from pyspark.sql.functions import udf
from pyspark.sql.types import IntegerType

# Criando sessão Spark
spark = SparkSession.builder.appName("UDFExample").getOrCreate()

# Criando DataFrame de exemplo
data = [(1,), (2,), (3,), (4,), (5,)]
df_spark = spark.createDataFrame(data, ["coluna_antiga"])

# Definindo uma função para dobrar os valores
def dobro(x):
    return x * 2

# Criando UDF (User Defined Function) com tipo de retorno Integer
dobro_udf = udf(dobro, IntegerType())

# Aplicando a UDF e criando uma nova coluna
df_spark = df_spark.withColumn("nova_coluna", dobro_udf(df_spark["coluna_antiga"]))

df_spark.show()
```

O que é um UDF (User Defined Function) em PySpark?

UDF (User Defined Function) é uma função personalizada criada pelo usuário para ser usada dentro de operações do PySpark.

O PySpark tem funções nativas de transformação (`col()`, `when()`, `lit()`, `regexp_replace()`, etc.), mas se você precisar de uma lógica mais complexa, pode usar UDFs.

Por que usar UDFs?

- Quando não há uma função nativa do PySpark para a transformação desejada.
- Para aplicar lógica mais complexa (ex: funções matemáticas personalizadas, processamento de strings, etc.).
- Para reutilizar código ao invés de escrever expressões complicadas em `.withColumn()`.

Cuidados ao Usar UDFs

- Mais lentas que funções nativas do PySpark (`when()`, `col()`, `regexp_replace()`, etc.), pois quebram a otimização do Catalyst.
- Sempre definir o tipo de retorno (ex: `IntegerType()`, `StringType()`), pois PySpark não infere automaticamente.
- Para melhor performance, use Pandas UDFs (`pandas_udf`) quando possível, pois são otimizadas com Arrow.

17. REGEX

O Regex (Expressão Regular) é uma técnica para filtrar, buscar e substituir padrões em strings

PANDAS

No Pandas, usamos `.str.contains()` com a opção `regex=True`.

```
Coluna_selecionada = df[df['nome_coluna'].str.contains('padrão', regex=True)]
```

```
# Exemplo
df_filtrado = df[df['nome'].str.contains(r'^A', regex=True)]
```

- O **regex ^A** significa "começa com A".
- O **str.contains(r'^A', regex=True)** filtra as linhas onde a coluna "nome" começa com A.

```
import pandas as pd

# Criando DataFrame de exemplo
data = {'nome': ['Alice', 'Bob', 'Carlos', 'Ana', 'Bárbara', 'Alberto']}
df = pd.DataFrame(data)

# Filtrar nomes que começam com "A"
df_filtrado = df[df['nome'].str.contains(r'^A', regex=True)]

print(df_filtrado)
```

PYSPARK

No PySpark, usamos a função `rlike()` para aplicar regex

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

# Criando sessão Spark
spark = SparkSession.builder.appName("RegexFilter").getOrCreate()

# Criando DataFrame
data = [("Alice",), ("Bob",), ("Carlos",), ("Ana",), ("Bárbara",), ("Alberto",)]
df_spark = spark.createDataFrame(data, ["nome"])

# Filtrar nomes que começam com "A"
df_filtrado = df_spark.filter(col("nome").rlike(r"^A"))

df_filtrado.show()
```

- **`rlike(r"^A")`** aplica um filtro onde o nome começa com A.

`regex_extract`

A função `regexp_extract()` no PySpark é usada para extrair parte de uma string que corresponde a um padrão regex (expressão regular)

Exemplo:

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import regexp_extract, col

# Criando sessão Spark
spark = SparkSession.builder.appName("RegexExtractExample").getOrCreate()

# Criando DataFrame de exemplo
data = [("Alice - ID: 1234",), ("Bob - ID: 5678",), ("Carlos - ID: 91011",)]
df_spark = spark.createDataFrame(data, ["info"])

# Extraíndo apenas o número do ID (dígitos após "ID:")
df_extraido = df_spark.withColumn("ID", regexp_extract(col("info"), r"ID: (\d+)", 1))

df_extraido.show()
```

`regexp_extract(col("info"), r"ID: (\d+)", 1)`

- O **regex `ID: (\d+)`** busca "ID: " seguido por um número.
- Os parênteses `()` capturam apenas os dígitos.
- O `1` indica que queremos extrair o primeiro grupo capturado.

Exemplos de Regex Úteis

Padrão	Regex	Explicação
Começa com "A"	<code>^A</code>	Filtra palavras que começam com "A".
Termina com "o"	<code>o\$</code>	Filtra palavras que terminam com "o".
Contém "ar"	<code>ar</code>	Filtra palavras que contêm "ar".
Apenas letras	<code>^[a-zA-Z]+\$</code>	Filtra apenas palavras sem números.
Apenas números	<code>^\d+\$</code>	Filtra apenas números.
E-mails	<code>^\S+@\S+\.\S+\$</code>	Filtra e-mails válidos.

Existe `regex_extract()` no Pandas?

Não existe um método chamado `regex_extract()` no Pandas. Mas podemos usar **`.str.extract()`**, que tem a mesma função.

```
1 import pandas as pd
2
3 # Criando DataFrame de exemplo
4 data = {"info": ["Alice - ID: 1234", "Bob - ID: 5678", "Carlos - ID: 91011"]}
5 df = pd.DataFrame(data)
6
7 # Extraíndo apenas o número do ID
8 df["ID"] = df["info"].str.extract(r"ID: (\d+)")
9
10 print(df)
```

- **`.str.extract(r"ID: (\d+)")`** funciona igual ao `regexp_extract()` do PySpark.
- O **`(\d+)`** captura apenas os números depois de **"ID:"**.

18. Window Function

PANDAS

O Pandas não tem window functions nativas como PySpark, mas conseguimos o mesmo comportamento com **.groupby()** e **.rolling()**

No exemplo, vamos criar um ranking por categoria:

```
# Exemplo: Criar Ranking por Categoria

import pandas as pd

# Criando DataFrame
df = pd.DataFrame({
    "categoria": ["A", "A", "A", "B", "B", "C"],
    "valor": [10, 20, 15, 40, 30, 25]
})

# Criando um ranking dentro de cada categoria
df["rank"] = df.groupby("categoria")["valor"].rank(method="dense", ascending=False)

print(df)
```

.groupby("categoria")["valor"].rank(method="dense", ascending=False)

- Cria um ranking dentro de cada categoria.
- O ranking é atribuído sem pular valores (method="dense").

Exemplo: Média Móvel

```
df["media_movel"] = df["valor"].rolling(window=2).mean()

print(df)
```

.rolling(window=2).mean()

calcula a média móvel com janela de 2 valores

PYSPARK

No PySpark, usamos a classe **Window** do módulo **pyspark.sql.window**

```
from pyspark.sql import SparkSession
from pyspark.sql.window import Window
from pyspark.sql.functions import rank, row_number, avg

# Criando sessão Spark
spark = SparkSession.builder.appName("WindowFunctionsExample").getOrCreate()

# Criando DataFrame
data = [
    ("A", 10), ("A", 20), ("A", 15),
    ("B", 40), ("B", 30), ("C", 25)
]
df_spark = spark.createDataFrame(data, ["categoria", "valor"])

# Criando Window (partitionBy "categoria" e orderBy "valor")
window_spec = Window.partitionBy("categoria").orderBy(df_spark.valor.desc())

# Aplicando Window function
df_spark = df_spark.withColumn("rank", rank().over(window_spec))
df_spark = df_spark.withColumn("row_number", row_number().over(window_spec))

df_spark.show()
```

- **Window.partitionBy("categoria").orderBy("valor.desc()")** : Define a partição e ordenação.
- **rank().over(window_spec)** : Gera um ranking por categoria, com empates.
- **row_number().over(window_spec)** : Gera um número único por linha, sem empates.

Exemplo de Média Móvel no PySpark

```
window_spec = Window.partitionBy("categoria").orderBy("valor").rowsBetween(-1, 0)

df_spark = df_spark.withColumn("media_movel", avg("valor").over(window_spec))

df_spark.show()
```

rowsBetween(-1, 0): Define a janela para calcular a média dos valores atuais e anteriores.

19. Exporte Data

PANDAS

Usamos `.to_csv()`, `.to_parquet()` e `.to_json()` para salvar os dados

```
# Exportar para CSV
df.to_csv("dados.csv", index=False, sep=";") # `index=False` remove índice

# Exportar para Parquet
df.to_parquet("dados.parquet", engine="pyarrow")

# Exportar para JSON
df.to_json("dados.json", orient="records", indent=4)
```

- **"dados.csv"**: Salva como CSV separado por ;.
- **"dados.parquet"**: Usa PyArrow para salvar como Parquet.
- **"dados.json"** : Salva como JSON no formato lista de dicionários (`orient="records"`).

No Pandas, ao salvar Parquet, o argumento `engine="pyarrow"` ou `engine="fastparquet"` define qual biblioteca será usada.

PYSPARK

No PySpark, usamos `.write.format().save()` ou métodos específicos.

```
# Exportar para CSV
df_spark.write.csv("dados_spark.csv", header=True, sep=";")

# Exportar para Parquet
df_spark.write.parquet("dados_spark.parquet")

# Exportar para JSON
df_spark.write.json("dados_spark.json")
```

Formato	Pandas	PySpark
CSV	<code>df.to_csv("file.csv", index=False, sep=";")</code>	<code>df_spark.write.csv("file.csv", header=True, sep=";")</code>
Parquet	<code>df.to_parquet("file.parquet")</code>	<code>df_spark.write.parquet("file.parquet")</code>
JSON	<code>df.to_json("file.json", orient="records", indent=4)</code>	<code>df_spark.write.json("file.json")</code>

20. Referências

- [Read CSV with pyspark - PysparkExamples](#)
- [Pyspark Documentations](#)
- [Pandas documentation](#)

Bons estudos!

