PySpark

PySpark, a powerful tool for processing and analyzing **big data**. It's tailored for **data engineers**, **data scientists**, and **ML enthusiasts**, focusing on working with **large-scale datasets in distributed environments**. The goal is to **transform raw data into insights**.

What is PySpark

- **Apache Spark**: An open-source **distributed computing system**.
- **PySpark**: The **Python API** for Spark, allowing parallel computation on large datasets.
- It's suitable for:
    - **Batch processing**
    - **Real-time streaming**
    - **Machine learning (ML)**
    - **SQL-based analytics**
- Industries benefiting from PySpark include **finance, healthcare, and e-commerce** due to its **speed and scalability**.

When to Use PySpark

PySpark shines when handling **big data that exceeds single-machine capacity**:

- **Distributed data processing** using Spark's in-memory architecture.
- **ML on large datasets** using Spark's **MLlib**.
- **ETL/ELT pipelines** for structured transformation of raw data.
- Works well with formats like **CSV, Parquet**, and others.

Spark Cluster
A **Spark cluster** consists of:

- **Master node**: Manages task distribution and resources.
- **Worker nodes**: Execute computing tasks. This setup allows **parallel, distributed processing** of large datasets, a core part of PySpark's power.

SparkSession

- **SparkSession** is the **entry point** for using PySpark.
- Enables access to Spark features like **SQL, streaming, ML, and data handling**.
- To create:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("MySparkApp").getOrCreate()
```

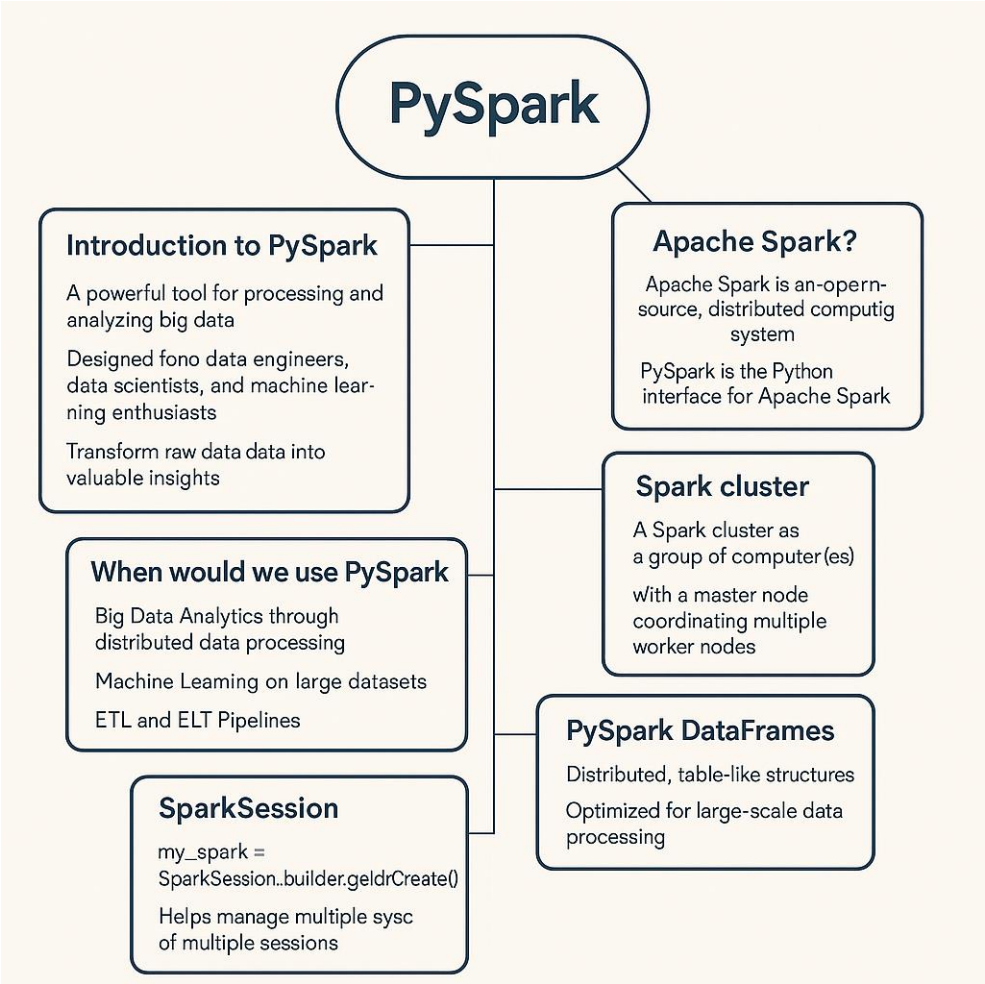
- The .appName() is useful for managing multiple applications.
- getOThis helps avoid creating **duplicate sessions**, which can waste memory or even crash the app.rCreate() ensures you don't accidentally start multiple sessions.

7. PySpark DataFrames

- Similar to **Pandas DataFrames**, but **distributed** for large-scale processing.
- Created using:

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

- df.show() displays contents.
- Syntax is intuitive for those familiar with Pandas, but underlying processing is optimized for **distributed environments**.



Introduction to PySpark DataFrames

PySpark DataFrames vs Pandas:

- **Pandas**: Operates on **a single machine** — not ideal for huge datasets.
- **PySpark**: Designed for **distributed computing** — data is spread across a cluster, enabling faster and scalable processing.

- **Why PySpark DataFrames matter**:
  - Efficient for big data handling.
  - Support common operations like **filtering, grouping, and aggregating**.
  - Provide **SQL-like syntax**, but execute operations differently behind the scenes.
- Although syntax might feel familiar to those with **Pandas experience**, the performance model is different due to **distributed execution**.

Creating DataFrames from Filestores

To read a CSV into a DataFrame:

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

- header=True: Treats the first row as column names.
- inferSchema=True: Automatically detects data types.
- Efficient for **loading structured data** into a distributed format.

Printing the DataFrame

To preview data:

df.show()

- Shows the **first 5 rows.**
- While you can also use createDataFrame(), read.csv() is **faster and better at scale**.

Printing DataFrame Schema

To view the structure (columns and data types):

df.printSchema()

This helps **understand the data** you're working with.

Basic Analytics on DataFrames

Common tasks:

- **Row count**:

df.count()

- **Group and aggregate**:

df.groupBy("gender").agg({"salary_usd": "avg"})

- Useful functions: sum(), avg(), stddev(), etc.
- Helpful for **summarizing big datasets** and spotting patterns.

Key Functions for Analytics

Key DataFrame methods:

Function	**Use Case**	**Equivalent in SQL**
select()	Choose specific columns	SELECT
filter()	Filter rows by condition	WHERE
groupBy()	Group rows by one or more columns	GROUP BY
agg()	Aggregate with a function like sum, avg	AGGREGATE

Key Functions Example

Example: Filter and select

df.filter(df.age > 50).select("name", "age").show()

- Filters rows where age > 50.
- Selects only the name and age columns.

🔷 **Takeaway:**

PySpark DataFrames are essential for **scalable, high-performance data processing**. They:

- Resemble Pandas in syntax but work across clusters.
- Support SQL-like operations.
- Allow flexible, powerful analytics on massive datasets.

Load the CSV file into a DataFrame

salaries_df = spark.read.csv("salaries.csv", header=True, inferSchema=True)

Count the total number of rows

row_count = salaries_df.count()
print(f"Total rows: {row_count}")

Group by company size and calculate the average of salaries

salaries_df.groupBy("company_size").agg({"salary_in_usd": "avg"}).show()
salaries_df.show()

Average salary for entry level in Canada

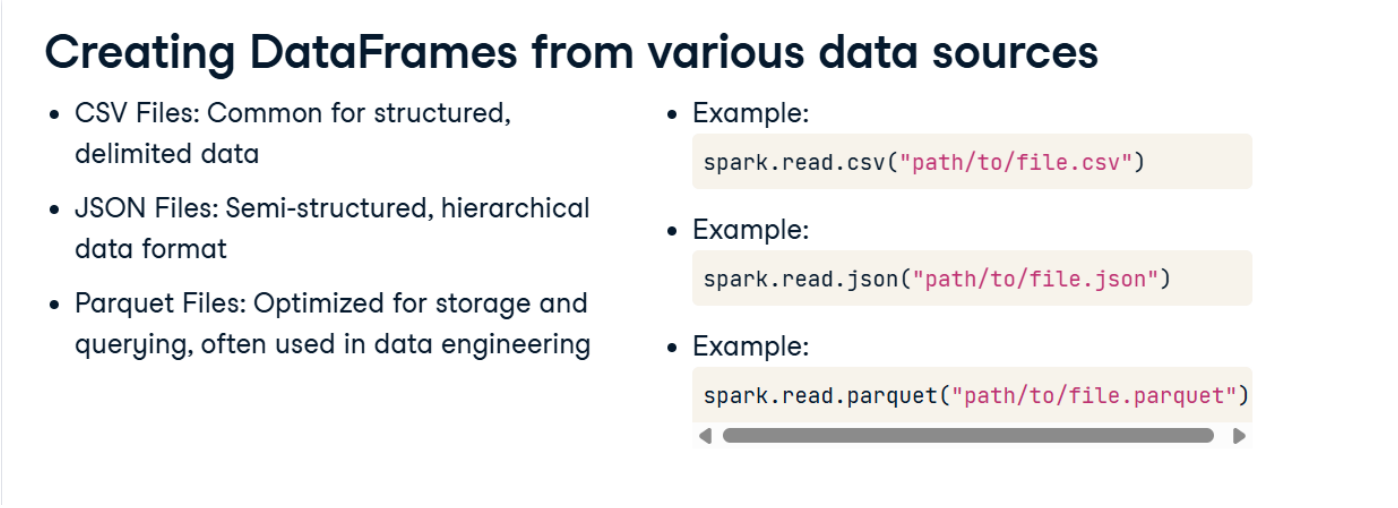
CA_jobs = ca_salaries_df.filter(ca_salaries_df['company_location'] == "CA").filter(ca_salaries_df['experience_level']

== "EN").groupBy().avg("salary_in_usd")

Show the result

CA_jobs.show()

Creating DataFrames from Various Data Sources



Schema Inference and Manual Schema Definition

Spark can automatically infer schemas, but it might misinterpret data types, particularly with complex or ambiguous data. Manually defining a schema can ensure accurate data handling.

DataTypes in PySpark DataFrames

To manually configure a schema, we define the datatype using the StructField function, calling the appropriate datatype method. PySpark DataFrames support various data types, similar to SQL and Pandas. Key types include:

DataTypes in PySpark DataFrames

- `IntegerType` : Whole numbers
  - E.g., `1` , `3478` , `-1890456`
- LongType: Larger whole numbers
  - E.g., 8-byte signed numbers, `922334775806`
- FloatType and DoubleType: Floating-point numbers for decimal values
  - E.g., `3.14159`
- StringType: Used for text or string data
  - E.g., `"This is an example of a string."`
- ...

DataTypes Syntax for PySpark DataFrames

We import specific classes from pyspark.sql.types. To define the schema for a DataFrame, we use StructType() and StructField() functions to define its structure and fields, filling in the columns and their types.

DataTypes Syntax for PySpark DataFrames

```
# Import the necessary types as classes
from pyspark.sql.types import (StructType,
                               StructField, IntegerType,
                               StringType, ArrayType)

# Construct the schema
schema = StructType([
    StructField("id", IntegerType(), True),
    StructField("name", StringType(), True),
    StructField("scores", ArrayType(IntegerType()), True)
])

# Set the schema
df = spark.createDataFrame(data, schema=schema)
```

from pyspark.sql.types import StructType, StructField, IntegerType, StringType

Fill in the schema with the columns you need from the exercise instructions

schema = StructType([StructField("age",IntegerType()),
StructField("education_num",IntegerType()),
StructField("marital_status",StringType()),
StructField("occupation",StringType()),
StructField("income",StringType()),
])

Read in the CSV, using the schema you defined above

census_adult = spark.read.csv("adult_reduced_100.csv",sep=',', header=False, schema=schema)

Print out the schema

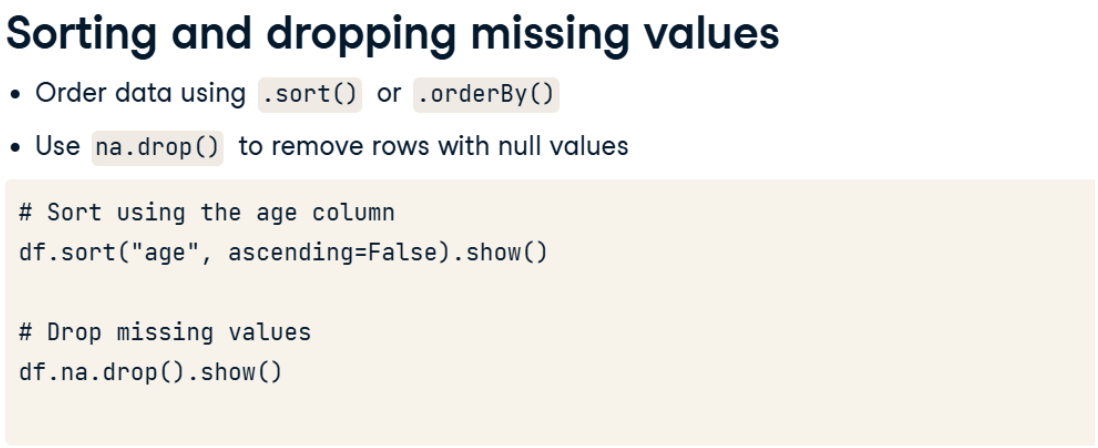
census_adult.printSchema()

DataFrame Operations - Selection and Filtering

Selecting specific columns and filtering rows are fundamental operations in data analysis. In Spark, these operations are efficient, even on large datasets, using methods like **.select(), .filter(), .sort(), and .where(). .**where() and .filter() work similarly to SQL, where we pass a column or columns and a condition to match.

Sorting and Dropping Missing Values

Sorting and handling missing values are common tasks. Dropping nulls can clean data, but sometimes we may want to fill or impute values instead, which Spark also supports. We can use .sort() for simple, flexible sorting and .orderby() for more complex, multi-column sorting. We can drop nulls using .na.drop().



Load the dataframe

census_df = spark.read.json("adults.json")

Filter rows based on age condition

salary_filtered_census = census_df.filter(census_df['age']>40)

Show the result

salary_filtered_census.show()

1. Handling Missing Data

Handling null values is crucial in data analysis as missing data can lead to skewed results or errors during processing. PySpark provides two primary methods for managing missing values:

- **Dropping Null Values**:
    - The .na.drop() method can be used to drop rows that contain null values.
    - This can be done across the entire DataFrame or for specific columns.
    - While this method simplifies the dataset, it might significantly reduce the dataset size if null values are common, which could lead to loss of important data.

Example:

df.na.drop() # Drops rows with nulls in any column

df.na.drop(subset=["column_name"]) # Drops rows with null in specific column

- **Replacing Null Values**:
    - The .na.fill() method is used to replace null values with a specified default value, ensuring the dataset remains complete and consistent.
    - This method is beneficial when null values are sparse, or when removing rows could result in significant data loss.

Example:

df.na.fill(value=0) # Replace all nulls with 0
df.na.fill({"column1": 0, "column2": "Unknown"}) # Replace nulls in specific columns

- **Column-Specific Filtering**:
    - You can also filter nulls in a specific column using .where() combined with .isNotNull() to retain rows where the column has non-null values.

Example:

df.filter(df["column_name"].isNotNull()) # Filter out rows with null in a specific column

- Use `.na.drop()` to remove rows with null values

```
# Drop rows with any nulls
df_cleaned = df.na.drop()

# Filter out nulls
df_cleaned = df.where(col("columnName").isNotNull())
```

- Use `.na.fill({"column": value)` to replace nulls with a specific value

```
# Fill nulls in the age column with the value 0
df_filled = df.na.fill({"age": 0})
```

2. Column Operations

PySpark provides several powerful methods for manipulating columns in a DataFrame:

- **Creating New Columns**:
    - The .withColumn() method allows for the creation of new columns based on expressions or transformations.
    - This is especially useful for derived metrics, such as calculating a new value from existing columns.

Example:

df = df.withColumn("new_column", df["existing_column"] * 2)
Create a new column with calculations

- **Renaming Columns**:
    - The .withColumnRenamed() method allows you to rename columns to make them more descriptive or standardized.
    - This improves the clarity of DataFrames, especially in collaborative settings.

Example:

df = df.withColumnRenamed("old_column_name", "new_column_name")

Renaming columns

- **Dropping Columns**:
    - The .drop() method is used to remove unnecessary columns from a DataFrame.
    - This is important for managing large datasets, improving memory usage, and focusing on relevant data.

Example:

df = df.drop("column_name") # Drop a column from the DataFrame

3. Row Operations

Row operations help in filtering, grouping, and aggregating data to analyze subsets or patterns within the data.

- **Filtering Rows**:
    - Filtering is used to narrow down the dataset based on specific conditions, such as selecting data from a particular date, region, or category.
    - The .filter() method applies a condition to the rows and returns only the rows that meet that condition.

Example:

df_filtered = df.filter(df ["age"] > 30) # Filter rows where age is greater than 30

- **Grouping Rows**:
    - Grouping allows you to organize data based on a field or category (e.g., by customer, product, or date).
    - This is crucial for analyzing patterns and trends within each group.
    - The .groupBy() method groups rows based on specified columns, and then you can perform aggregation on those groups.

Example:

df_grouped = df.groupby("category").agg({"value_column": "sum"}) # Group by category and sum the value column

- **Aggregation**:
    - After grouping, you can apply aggregation functions like sum(), avg(), count(), etc., to get summarized insights.
    - PySpark supports various aggregation functions which can be used after grouping the data.

Example:

df_grouped = df.groupby("category").agg({"value_column": "avg"}) # Group by category and calculate the average of a column

Summary:

- **Handling Missing Data**: PySpark offers .na.drop() and .na.fill() for dealing with null values. .drop() removes rows with nulls, and .fill() replaces nulls with a default value.
- **Column Operations**: Use .withColumn() to create new columns, .withColumnRenamed() to rename columns, and .drop() to remove unnecessary columns.
- **Row Operations**: .filter() allows filtering rows based on conditions, and .groupBy() followed by aggregation functions helps in grouping and analyzing data efficiently.

These operations provide flexibility to clean, manipulate, and transform data in PySpark, making it easier to analyze and extract valuable insights from large datasets.

Joins in PySpark

Joins in PySpark are used to combine data from multiple DataFrames based on shared columns, similar to SQL join operations. Joins are essential for enriching datasets by merging information from different sources.

- **Types of Joins**:
    - **Inner Join**: Returns only the rows with matching values in both DataFrames.
    - **Left Join (Left Outer Join)**: Returns all rows from the left DataFrame, with matching rows from the right DataFrame; if no match, returns null for the right DataFrame.
    - **Right Join (Right Outer Join)**: Returns all rows from the right DataFrame, with matching rows from the left DataFrame; if no match, returns null for the left DataFrame.
    - **Full Outer Join**: Returns all rows when there is a match in one of the DataFrames, with null values where there is no match.
- **Syntax**: The **.join()** method is used to perform the join operation. You specify the second DataFrame, the join type, and the columns on which to join the DataFrames.
    - If the columns have different names, you can explicitly specify the joining columns.

Example:

```
df1.join(df2, df1["column1"] == df2["column2"], "inner")
```

Union Operation

The **union** operation in PySpark combines or "stacks" two DataFrames with the same schema (same number of columns, with matching types and order). This operation is crucial when datasets are split across different sources or time periods and need to be consolidated into one.

- **Conditions for Union**:
    - The DataFrames must have the same number of columns and matching data types for the operation to succeed. If they don't match, PySpark will throw an error.
    - The union operation is typically used to combine data from separate files, such as monthly sales data into a unified yearly dataset.

```
df_union = df1.union(df2)
```

    - This stacks df2 below df1, creating a single DataFrame that consolidates rows from both DataFrames.

- **Schema Alignment**: It is critical to ensure that the schemas of the two DataFrames align correctly. Mismatched schemas or column types will prevent the union from working.

Working with Arrays and Maps

Complex data types like **arrays** and **maps** in PySpark provide the flexibility to handle more intricate data structures within DataFrames.

- **Arrays**: Arrays allow for the storage of multiple values in a single column, useful for attributes that have multiple values. You can define an array column using the appropriate data type.

Example:

```
from pyspark.sql.functions import lit

df = df.withColumn("array_column", lit([1, 2, 3]))
```

- **Maps**: Maps store dynamic key-value pairs, making them ideal for situations where the attributes of a row vary. Each row can have different keys in a map, offering flexible data storage.

Example:

```
from pyspark.sql.functions import map

df = df.withColumn("map_column", map(lit("key1"), lit("value1"), lit("key2"), lit("value2")))
```

Note: The map() method allows dynamic assignment of key-value pairs.

Working with Structs

Structs in PySpark group related fields together within a single column. This is useful for handling hierarchical or nested data, where you want to store multiple fields in one column but maintain the structure.

- **Structs** are defined using StructField, where each field has a name and data type.
- This allows you to manage nested relationships within a DataFrame more efficiently.

Example:

```
from pyspark.sql.types import StructType, StructField, StringType
struct_type = StructType([StructField("name", StringType(), True), StructField("age", StringType(), True)])
```

Summary:

- **Joins**: PySpark supports several join types (inner, left, right, and full outer) to combine datasets based on shared columns, similar to SQL.
- **Union**: The union operation allows you to stack two DataFrames with the same schema, combining their rows into a single dataset.
- **Arrays and Maps**: PySpark's complex data types, such as arrays and maps, offer flexibility in managing hierarchical or dynamic data within a column.
- **Structs**: Structs allow grouping related fields within a single column, helping to manage complex data structures in PySpark.

These advanced DataFrame operations are crucial for efficiently handling and manipulating complex datasets in PySpark, providing the ability to perform sophisticated analyses and data transformations.

- Examine the airports DataFrame. Note which key column will let you join airports to the flights table.
- Join the flights with the airports DataFrame on the "dest" column. Save the result as flights_with_airports.
- Examine flights_with_airports again. Note the new information that has been added.

```
# Examine the data
airports.show()

# .withColumnRenamed() renames the "faa" column to "dest"
airports = airports.withColumnRenamed("faa", "dest")

# Join the DataFrames
flights_with_airports = flights.join(airports, on='dest', how='left outer')
```

1. **Introduction to UDFs**
    - UDF stands for **User-Defined Function** in PySpark.
    - UDFs are custom functions that you define and use within PySpark DataFrames to manipulate data in ways that built-in PySpark functions may not support.
    - There are two main types of UDFs in PySpark: **PySpark UDFs** and **pandas UDFs**.
2. **UDFs for Repeatable Tasks**
    - UDFs are designed to be **reusable** and **repeatable**. This means that once created and registered with the SparkSession, they can be applied multiple times across different DataFrames.
    - **PySpark UDFs** are suitable for handling smaller datasets.
    - **pandas UDFs** are designed for **larger datasets** and typically offer better performance in those cases.
    - Despite differences in performance, both types of UDFs operate similarly, but the performance optimization comes from how they handle data row by row.
3. **Defining and Registering a UDF**
    - To create a UDF, you first define a standard Python function. For example, a function like to_upper_case() could be used to convert text data in a column to uppercase.
    - Once you define the function, you need to **register it** as a UDF using the .udf() function in PySpark and specify the correct data type (e.g., StringType() for string operations).
    - **Registration** is important because it makes the UDF accessible across all worker nodes in the Spark cluster through the SparkSession.

Example code:

```
from pyspark.sql.functions import udf
from pyspark.sql.types import StringType
def to_upper_case(input_string):
    return input_string.upper()


# Register the UDF
upper_case_udf = udf(to_upper_case, StringType())
```

Apply the UDF to a DataFrame

df = df.withColumn("upper_case_column", upper_case_udf(df["column_name"]))
df.show()

Defining and registering a UDF

All PySpark UDFs need to be registered via the `udf()` function.

```
# Define the function
def to_uppercase(s):
    return s.upper() if s else None

# Register the function
to_uppercase_udf = udf(to_uppercase, StringType())

# Apply the UDF to the DataFrame
df = df.withColumn("name_upper", to_uppercase_udf(df["name"]))

# See the results
df.show()
```

4. **pandas UDF**
    - **pandas UDFs** are designed to work with **larger datasets** and provide better performance.
    - You define a pandas UDF using the @pandas_udf() decorator and specify the return type (e.g., @pandas_udf("float")).
    - Unlike PySpark UDFs, pandas UDFs do not need to be explicitly registered with the SparkSession as they are more efficient at handling data in bulk.

Example code:

from pyspark.sql.functions import pandas_udf

import pandas as pd

Define a pandas UDF

@pandas_udf("float")

def calculate_square(s: pd.Series) -> pd.Series:

**   return s * s**

Apply the pandas UDF to a DataFrame

df = df.withColumn("squared_column", calculate_square(df["numeric_column"]))

df.show()

pandas UDF

- Eliminates costly conversions of code and data
- Does not need to be registered to the SparkSession
- Uses pandas capabilities on extremely large datasets

```
from pyspark.sql.functions import pandas_udf

@pandas_udf("float")
def fahrenheit_to_celsius_pandas(temp_f):
    return (temp_f - 32) * 5.0/9.0
```

5. **PySpark UDFs vs. pandas UDFs**
    - **PySpark UDFs** are ideal for **small datasets** and **simple transformations**. They operate at the column level and require registration with the SparkSession.
    - **pandas UDFs** are better for **large datasets** because they leverage the power of **pandas**, making them more efficient for processing large volumes of data.
    - The decision to use one over the other depends on several factors, including:
        - **Data size**: Small datasets work better with PySpark UDFs, while large datasets benefit from pandas UDFs.
        - **Complexity**: Simple transformations may be fine with PySpark UDFs, but for more complex operations on large data, pandas UDFs offer performance benefits.
        - **Performance considerations**: pandas UDFs typically offer better performance, but they may also require a more complex setup and handling of data types.

Define a Pandas UDF that adds 10 to each element in a vectorized way
@pandas_udf(DoubleType())
def add_ten_pandas(column):
    return column + 10

Apply the UDF and show the result

df.withColumn("10_plus", add_ten_pandas(df['value']))
df.show()