Day 3 | PySpark Transformations Deep Dive

What is Pandas?

Pandas is a Python library used for data analysis on single machines.

Key characteristics :

- *Works in-memory (RAM)*
- *Best for small to medium datasets*
- *Very easy to learn & use*
- *Mostly used in:*

->Jupyter notebooks

->Data cleaning

->Exploratory Data Analysis (EDA)

Example:

```
import pandas as pd

df = pd.read_csv("sales.csv")
df.groupby("category")
["revenue"].sum()
```

What is PySpark?

PySpark is the Python API for Apache Spark, designed for big data processing.

Key characteristics :

- *Works in distributed systems*
- *Handles large datasets (GBs–TBs)*
- *Uses clusters instead of one machine*
- *Optimized for performance & scalability*
- *Used in platforms like Databricks*

Example:

```
from pyspark.sql import functions as F

df = spark.read.csv("sales.csv",
header=True)
df.groupBy("category").agg(F.sum("reve
nue"))
```

Pandas vs PySpark (Quick Comparison)

Feature	Pandas	PySpark
Best For	Small to medium-sized datasets	Large-scale data processing
Performance	Single-threaded, runs in memory	Distributed, optimized for big data
Scalability	Limited to available RAM	Scales across multiple nodes in a cluster
Ease of Use	Simple, Pythonic API	Requires understanding of distributed computing
Operations	Fast for small data, struggles with large data	Optimized for parallel execution
Integration	Works well with Python ecosystem	Works with Hadoop, Spark, and cloud platforms
Integration	Works well with Python ecosystem	Works with Hadoop, Spark, and cloud platforms

When should you use Pandas?

Use Pandas when:

- *Dataset fits in RAM*
- *You want quick analysis*
- *Doing EDA or prototyping*
- *Working locally on your laptop*

When should you use PySpark?

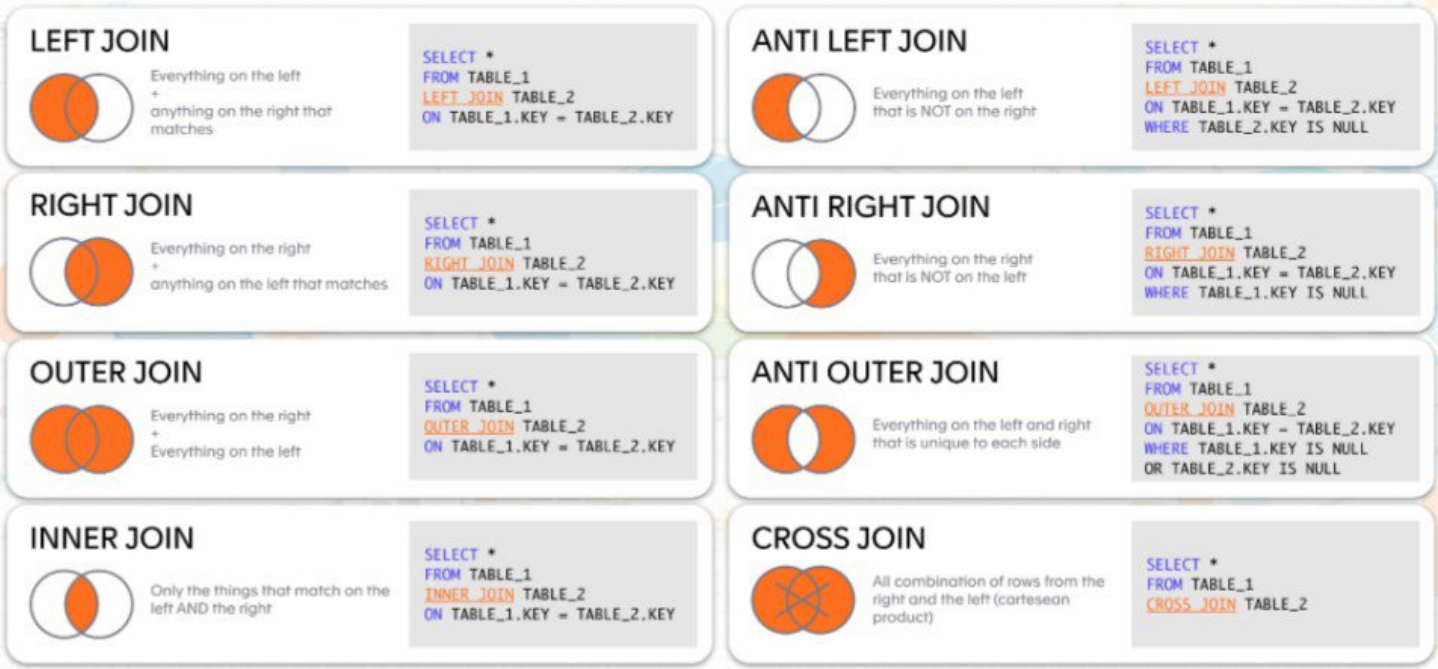
Use PySpark when:

- *Dataset is very large*
- *Data comes from cloud / data lake*
- *You need scalability*
- *Working in Databricks*
- *Running joins, window functions, pipelines*

🔗 What are Joins?

Joins are used to combine data from two tables/DataFrames based on a common column.

👉 *Very common in real-world data (orders + customers, sales + products, etc.)*



1️⃣ Inner Join

👉 *Only matching records from both tables*

✔ *Rule:*
Keep rows where key exists in both tables

Example :

```
df_customers.join(df_orders,
"customer_id", "inner")
```

Result:

customer_id	name	amount
1	Asha	500
2	Ravi	300

2️⃣ Left Join

👉 *All rows from LEFT table + matching from RIGHT*

✔ *Rule:*
If no match → null

Example :

```
df_customers.join(df_orders,
"customer_id", "left")
```

Result:

customer_id	name	amount
1	Asha	500
2	Ravi	300
4	null	700

3️⃣ Right Join

👉 *All rows from RIGHT table + matching from LEFT*

Example :

```
df_customers.join(df_orders,
"customer_id", "right")
```

Result:

customer_id	name	amount
1	Asha	500
2	Ravi	300
4	null	700

4️⃣ Full Outer Join

👉 *All rows from BOTH tables*

Example :

```
df_customers.join(df_orders,
"customer_id", "outer")
```

Result:

customer_id	name	amount
1	Asha	500
2	Ravi	300
3	Neha	null
4	null	700

Easy Memory Trick

Join Type	Meaning
Inner	Common data
Left	Keep left table
Right	Keep right table
Outer	Keep everything

🪟 What are Window Functions?

Window functions perform calculations across a group of rows
👉 without collapsing rows (unlike groupBy).

🔑 This is why they are super powerful for:
->Running totals
->Rankings
->Moving averages

🤔 GroupBy vs Window

❌ *groupBy*

->Aggregates data

->Reduces number of rows

✅ *Window*

->Keeps all rows

->Adds new calculated columns

📌 Sample Data (Sales)

date	category	revenue
01-01	A	100
02-01	A	200
03-01	A	300
01-01	B	50
02-01	B	150

🧩 Window Function Structure

```
from pyspark.sql.window import Window
from pyspark.sql import functions as F

window_spec =
Window.partitionBy("category").orderBy
("date")
```

🔹 *partitionBy("category")*
→ *Groups rows by category (like groupBy, but keeps all rows)*
🔹 *orderBy("date")*
→ *Sorts data by date inside each category*
🔹 *window_spec*
→ *Defines how window calculations (running total, rank) should work*
👉 *Used with .over(window_spec) to calculate values without collapsing rows.*

1️⃣ Running Total (Cumulative Sum)

🎯 *Goal:*

Total revenue till that date per category

📌 **Result:**

date	category	revenue	running_total
01-01	A	100	100
02-01	A	200	300
03-01	A	300	600
01-01	B	50	50
02-01	B	150	200

2 Ranking (row_number, rank, dense_rank)

- *row_number()*

Gives unique row number

```
Window.partitionBy("category").orderBy
(F.desc("revenue"))
```

```
df.withColumn("row_num",
F.row_number().over(window_spec))
```

2 Ranking (row_number, rank, dense_rank)

- *rank()*

•*Same values → same rank*
•*Skips numbers*

```
df.withColumn("rank",
F.rank().over(window_spec))
```

- *dense_rank()*

•*Same values → same rank*
•*No gaps*

```
df.withColumn("dense_rank",
F.dense_rank().over(window_spec))
```

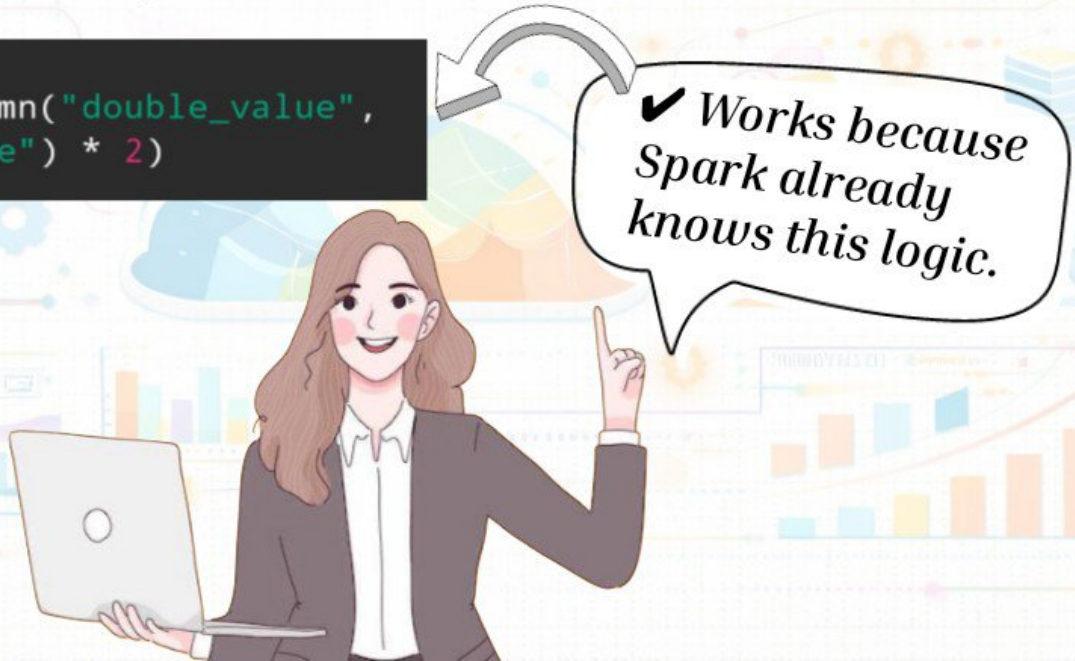
🧩 What is a UDF?

A User-Defined Function (UDF) lets you apply custom Python logic to PySpark DataFrame columns.

👉 *Use UDFs when built-in Spark functions are not enough.*

📌 *Simple Example (Without UDF)*

```
df.withColumn("double_value",
F.col("value") * 2)
```



📌 When do we need a UDF?

When logic is custom, like:

- *Categorizing values*
- *Custom business rules*
- *String manipulation not available in Spark*

UDF Example

🎯 Goal: Label revenue as High or Low

```
from pyspark.sql.functions import udf
from pyspark.sql.types import
StringType

def revenue_label(amount):
    if amount >= 200:
        return "High"
    else:
        return "Low"

label_udf = udf(revenue_label,
StringType())

df = df.withColumn("revenue_type",
label_udf(df.revenue))
```

🧠 What's happening?

Python function → revenue_label
Converted to Spark UDF → label_udf
Applied column-wise to DataFrame

⚠️ Important Warning

❌ UDFs are slower than Spark built-in functions
✔ Always try Spark functions first
Use UDF only when necessary.

🔍 What I Learned Today

✅Pandas vs PySpark – when to use what
✅Joins – inner, left, right & outer
✅Window functions – running totals & rankings
✅User-Defined Functions (UDFs)

🛠️ Skills Strengthened

🧠 Thinking in distributed data
✍️ Writing efficient PySpark transformations
🔍 Solving real-world data problems

💡 Key Takeaway

“Big data needs the right tools — PySpark brings scalability, speed, and structure.”