DataFrame Operations

PART- II

CONDITIONAL LOGIC IN SPARK

- Conditional logic allows us to apply **if-else condition** on DataFrame columns to derive new columns or manipulate existing one
- Spark Provides this using **when()** and **otherwise()** from **pyspark.sql.function**
- It is equivalent to CASE WHEN in SQL.

SQL

```
SELECT
  name,
  salary,
CASE
   WHEN salary > 50000 THEN 'High'
   WHEN salary > 30000 THEN 'Medium'
   ELSE 'Low'
END AS salary_band
FROM employees;
```

Pyspark

```
from pyspark.sql.functions import when, col

df = df.withColumn( "salary_band",
when(col("salary") > 50000, "High")
.when(col("salary") > 30000, "Medium")
.otherwise("Low")
)
```

Note:

- In PySpark, the `when` function is part of the **`pyspark.sql.functions`** module and can be imported directly. **otherwise() is not a standalone function- It's a method of the object returned by when()**
- importing otherwise directly will throw an error (as otherwise() is not a function available in **`pyspark.sql.functions`**

📘 HANDLING NULL VALUES IN SPARK DATAFRAME

✅ 1. Detecting Nulls:

Use **isNull()** or **isNotNull()** functions.

```
from pyspark.sql.functions import col
#Filter rows where ‘age’ is null
df.filter(col("age").isNull()).show()
#Filter rows where ‘age’ is not null
df.filter(col("age").isNotNull()).show()
```

✅ 2. Dropping Nulls:

Use **dropna()**

```
#Drop rows with any null value
df.dropna().show()
#Drop rows where all columns are null
df.dropna(how='all').show()
#Drop rows if 'age' or 'salary' is null
df.dropna(subset=['age', 'salary']).show()
```

✅ 3. Filling Nulls:

Use **fillna()** to replace nulls with a specific value.

```
#Fill all nulls with zero
df.fillna(0).show()
#Fill nulls in specific columns
df.fillna({'age': 25, 'salary': 50000}).show()
```

✅ 4. Replacing Nulls with Values from Other Columns:

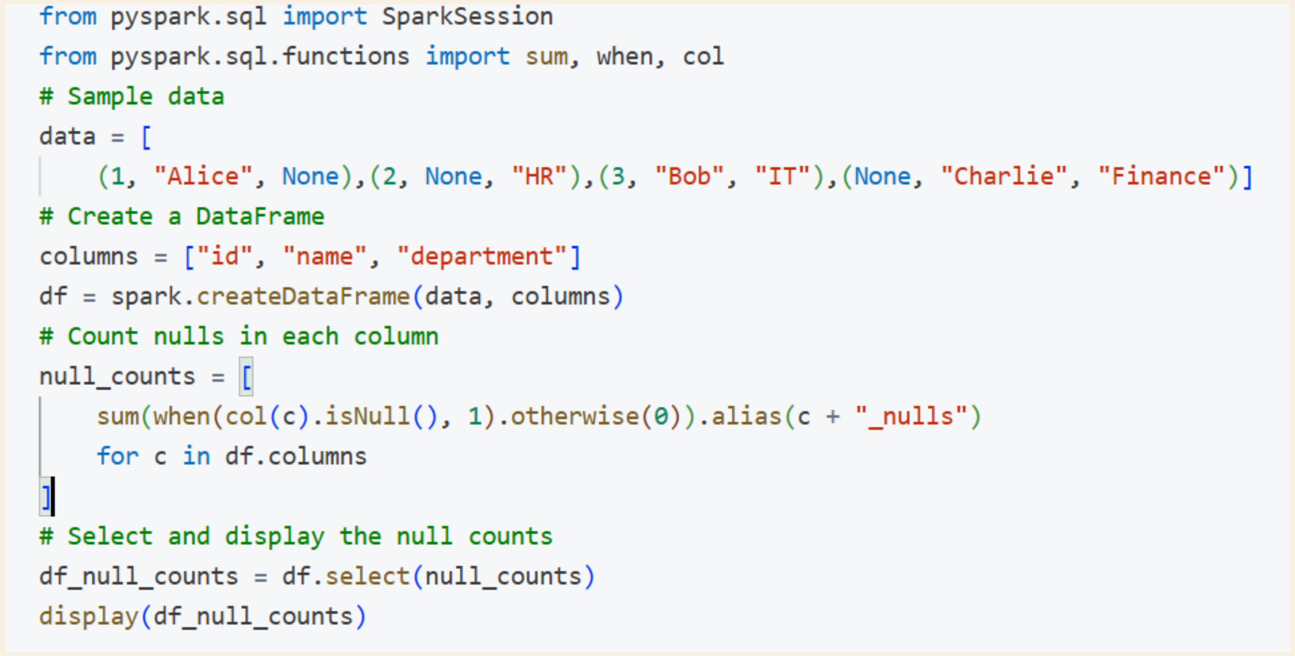
Using **when()** and **otherwise()** .

```
from pyspark.sql.functions import when
df.withColumn("final_salary",when(col("salary").isNull(), col
("expected_salary")).otherwise(col("salary"))).show()
```

HANDLING NULL VALUES IN SPARK DATAFRAME

✅ 5. Counting Nulls:

To count nulls per column:



```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import sum, when, col
# Sample data
data = [
    (1, "Alice", None),(2, None, "HR"),(3, "Bob", "IT"),(None, "Charlie", "Finance")]
# Create a DataFrame
columns = ["id", "name", "department"]
df = spark.createDataFrame(data, columns)
# Count nulls in each column
null_counts = [
    sum(when(col(c).isNull(), 1).otherwise(0)).alias(c + "_nulls")
    for c in df.columns
]
# Select and display the null counts
df_null_counts = df.select(null_counts)
display(df_null_counts)
```

Output

id_nulls	name_nulls	department_nulls
1	1	1

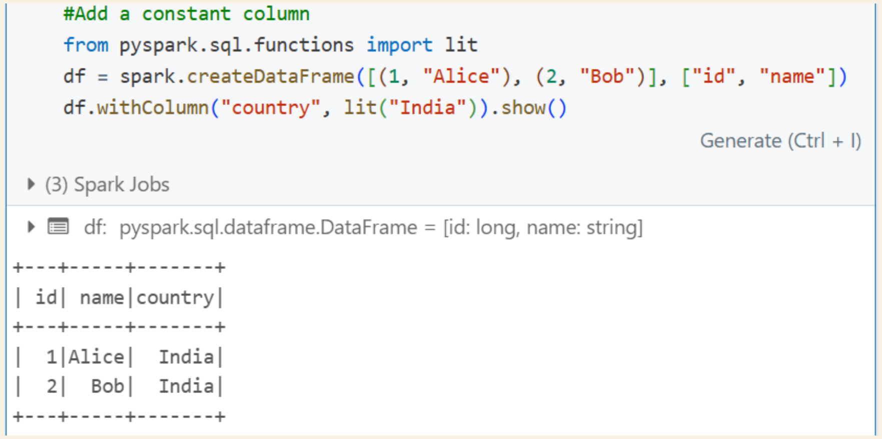
Best Practices:

- ✓ Prefer **dropna()** only when data volume is high and nulls are sparse.
- ✓ Use fillna() with domain-specific default values.
- ✓ Always explore null distribution before applying fixes.

LIT() FUNCTION IN PYSPARK

📌 What is lit()?

The **lit()** function is used to add a **literal (constant)** value to a PySpark DataFrame column. It's part of **pyspark.sql.functions.**



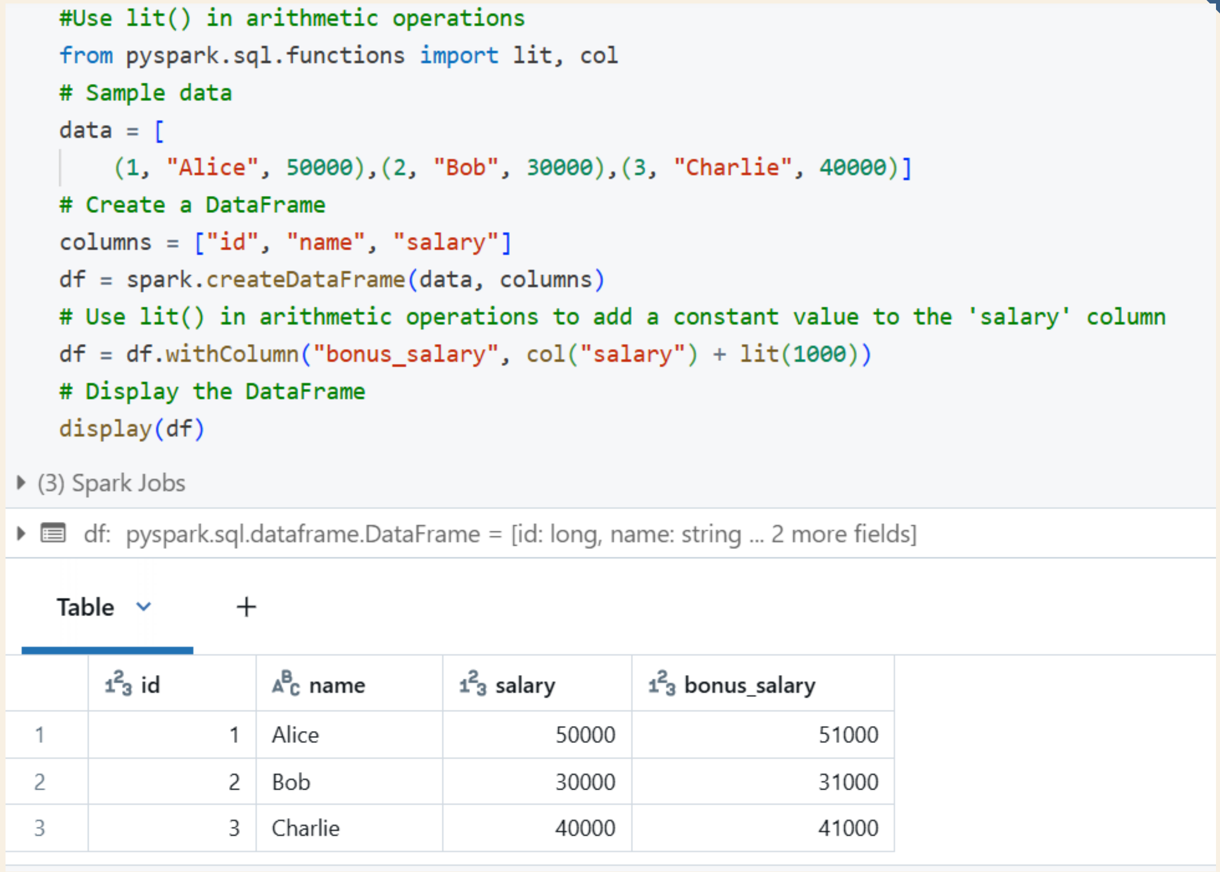
```
#Add a constant column
from pyspark.sql.functions import lit
df = spark.createDataFrame([(1, "Alice"), (2, "Bob")], ["id", "name"])
df.withColumn("country", lit("India")).show()
```

Generate (Ctrl + I)

(3) Spark Jobs

df: pyspark.sql.dataframe.DataFrame = [id: long, name: string]

```
+---+-----+-------+
| id| name|country|
+---+-----+-------+
|  1|Alice|  India|
|  2|  Bob|  India|
+---+-----+-------+
```



```
#Use lit() in arithmetic operations
from pyspark.sql.functions import lit, col
# Sample data
data = [
    (1, "Alice", 50000),(2, "Bob", 30000),(3, "Charlie", 40000)]
# Create a DataFrame
columns = ["id", "name", "salary"]
df = spark.createDataFrame(data, columns)
# Use lit() in arithmetic operations to add a constant value to the 'salary' column
df = df.withColumn("bonus_salary", col("salary") + lit(1000))
# Display the DataFrame
display(df)
```

(3) Spark Jobs

df: pyspark.sql.dataframe.DataFrame = [id: long, name: string ... 2 more fields]

Table

	id	name	salary	bonus_salary
1	1	Alice	50000	51000
2	2	Bob	30000	31000
3	3	Charlie	40000	41000

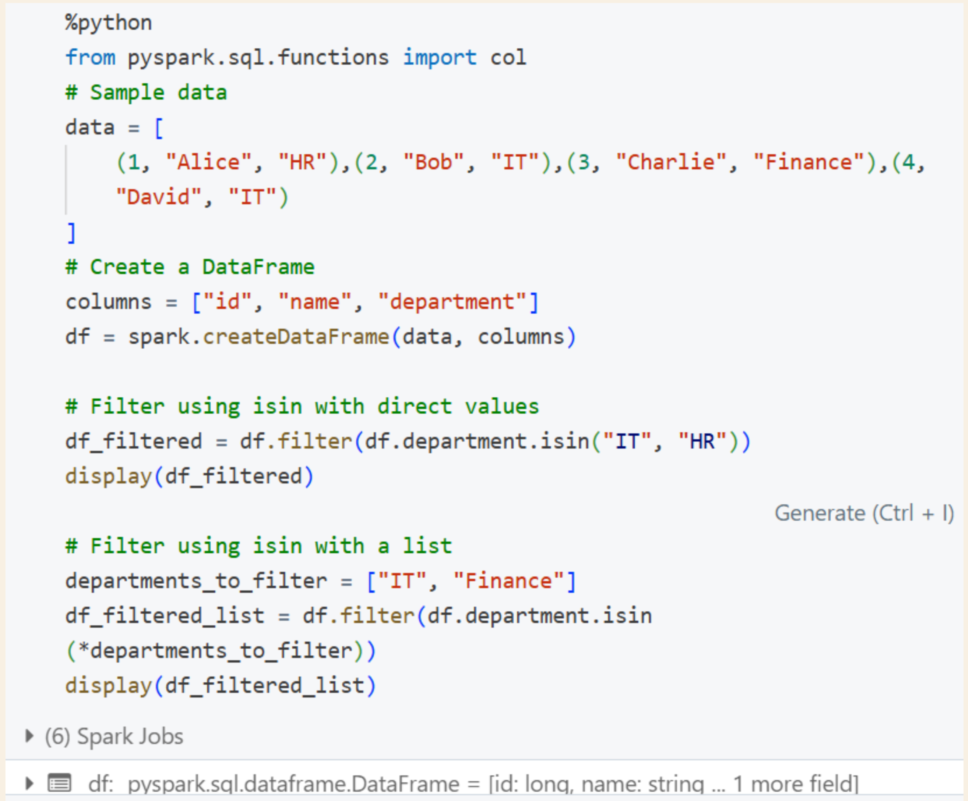
Note: Spark expects all column values to be Column objects, and constants must be wrapped with lit() to conform.

It’s required when combining a constant with column values in transformations.

isin() function in pyspark

The **isin()** function is used to filter rows where the column's value is in a given list of values.

It works like **SQL's IN (...).**

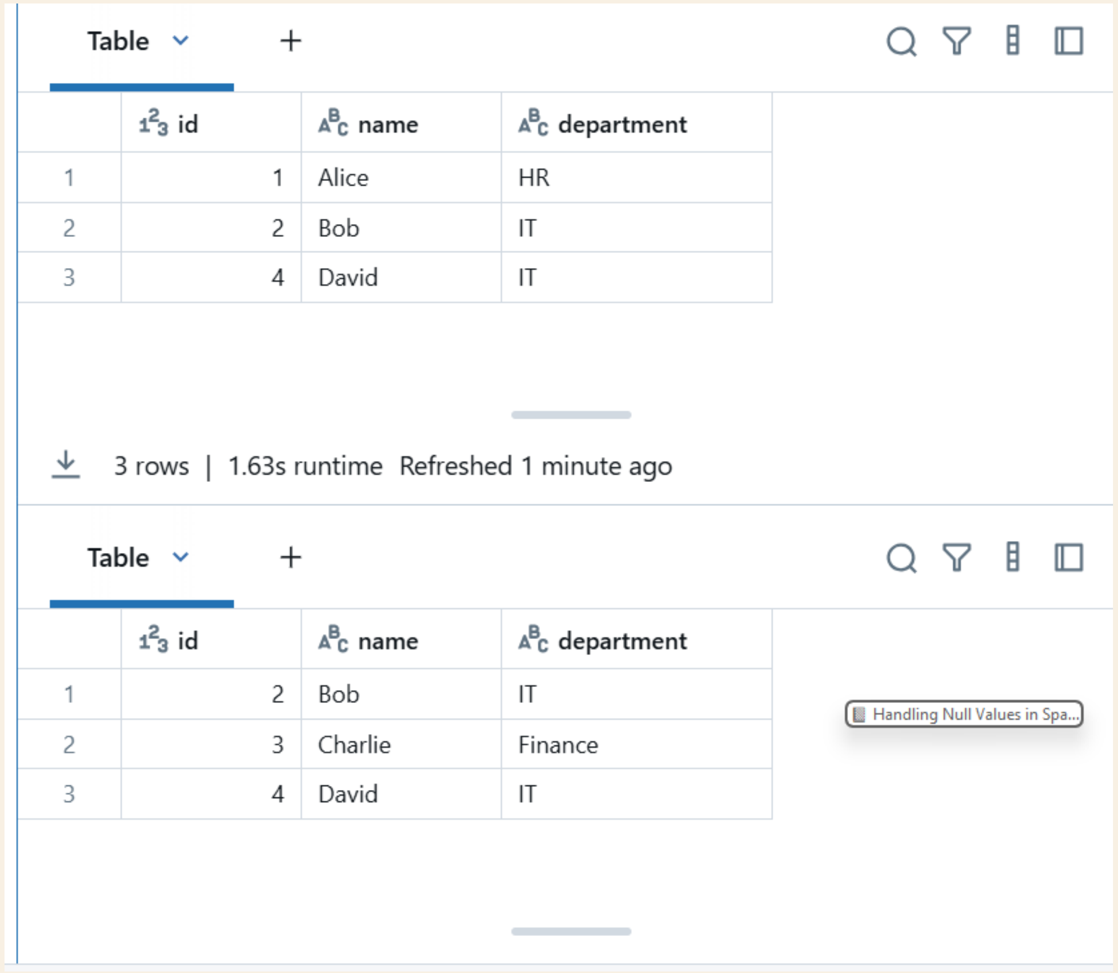


```
%python
from pyspark.sql.functions import col
# Sample data
data = [
    (1, "Alice", "HR"),(2, "Bob", "IT"),(3, "Charlie", "Finance"),(4,
    "David", "IT")
]
# Create a DataFrame
columns = ["id", "name", "department"]
df = spark.createDataFrame(data, columns)

# Filter using isin with direct values
df_filtered = df.filter(df.department.isin("IT", "HR"))
display(df_filtered)
                                                    Generate (Ctrl + I)
# Filter using isin with a list
departments_to_filter = ["IT", "Finance"]
df_filtered_list = df.filter(df.department.isin
(*departments_to_filter))
display(df_filtered_list)
```

```
▸ (6) Spark Jobs
▸ df:  pyspark.sql.dataframe.DataFrame = [id: long, name: string ... 1 more field]
```

output



	id	name	department
1	1	Alice	HR
2	2	Bob	IT
3	4	David	IT

3 rows | 1.63s runtime Refreshed 1 minute ago

	id	name	department
1	2	Bob	IT
2	3	Charlie	Finance
3	4	David	IT

Note:

- You must use * before a Python list to unpack it into arguments.
- Works on strings, integers, etc.
- It's useful in filter, where, and even with when conditions.