09 DataFrame API Transformations

Enforcing schema :

```
# method 1: creating DDL schema
orders_schema = 'order_id int,customer_id int,product_id int,unit_price floa
t,order_date date,order_status string,state string,quantity int'
df = spark.read.format("csv").schema(orders_schema).option("header" , "t
rue").load("/data/orders_40mb.csv")


# method 2 : creating schema using StructType and StructField
# https://spark.apache.org/docs/latest/sql-ref-datatypes.html
from pyspark.sql.types import  StructType, StructField , IntegerType, String
Type, DateType, FloatType
orders_schema = StructType([
StructField("order_id",IntegerType(), False),
StructField("customer_id",IntegerType(), True),
StructField("product_id",IntegerType(), True),
StructField("unit_price",FloatType(), True),
StructField("order_date",DateType(), True),
StructField("order_status",StringType(), True),
StructField("state",StringType(), True),
StructField("quantity",IntegerType(), True)
])

df = spark.read.format("csv").schema(orders_schema).option("header" , "t
rue").load("/data/orders_40mb.csv")
# sortcut method
df = spark.read.csv("path", header = True , schema = orders_schema)
```

Spark read modes:

Spark controls how it behaves when it encounters **bad or malformed records** using `mode`.

Default mode is **PERMISSIVE**.

📘 PERMISSIVE

Loads all records and replaces malformed fields with null values.

Default mode and safest option for exploratory ingestion.

Example: `spark.read.option("mode","PERMISSIVE").csv(path)`

📘 DROPMALFORMED

Drops rows that do not match the schema.

Useful when bad records can be ignored but causes silent data loss.

Example: `spark.read.option("mode","DROPMALFORMED").csv(path)`

📘 FAILFAST

Stops the job immediately when a bad record is encountered.

Best for production pipelines with strict data quality rules.

Example: `spark.read.option("mode","FAILFAST").csv(path)`

```
orders_schema = 'order_id int,customer_id int,product_id int,unit_price floa
t,order_date date,order_status string,state string,quantity int'
# default is PERMISSIVE mode

df = spark.read.format("csv").schema(orders_schema).option("header" , "t
rue").load("/data/orders_40mb.csv")

df = spark.read.format("csv").schema(orders_schema).option("header" , "t
rue").option("mode","DROPMALFORMED").load("/data/orders_40mb.csv")

df = spark.read.format("csv").schema(orders_schema).option("header" , "t
```

```
rue").option("mode","FAILFAST").load("/data/orders_40mb.csv")

df.show()

# store corrrupted data
# create a column in the schema with name set by below property
spark.conf.get("spark.sql.columnNameOfCorruptRecord")
(default value :  _corrupt_record)

orders_schema = 'order_id int,customer_id int,product_id int,unit_price floa
t,order_date date,order_status string,state string,quantity int, _corrupt_reco
rd string'

df = spark.read.format("csv").schema(orders_schema).option("header" , "t
rue").load("/data/orders_40mb.csv")
df.show(truncate=False)
```

Different ways to create a dataframe:

```
spark.read()
spark.sql()
spark.table()
spark.range(5)
spark.range(0,9)
spark.range(0,10,2)

# using createDataFrame function

data_list = [
(1,'2025-01-01',100 , 'Fashion')
,(2,'2025-01-02',200 , 'Electronics')
,(3,'2025-01-01',300 , 'Electronics')
,(4,'2025-01-03',400 , 'Fashion')
]
spark.createDataFrame(data_list)
```

```
table_schema = 'order_id int , order_date string, sales int, category string'

df = spark.createDataFrame(data_list , schema = table_schema)

from datetime import date

data_list = [
(1,date(2025,1,11),100 , 'Fashion')
,(2,date(2025,1,11),200 , 'Electronics')
,(3,date(2025,1,11),300 , 'Electronics')
,(4,date(2025,1,11),400 , 'Fashion')
]
table_schema = 'order_id int,order_date date,sales int,category string'
df = spark.createDataFrame(data_list , schema = table_schema)
df.show()

# create dataframe infer schema by default
table_schema = ["order_id" ,"order_date" ,"sales" , "category"]
df = spark.createDataFrame(data_list , schema = table_schema)
df.printSchema()
```

Different ways to select columns in dataframe

```
# string method
df.select("order_id","unit_price", "order_status", "quantity").show()

df.select("order_id","unit_price", "order_status", expr("quantity+2 as q")).sh
ow()
df.select("order_id","unit_price", "order_status", expr("quantity*unit_price a
s sales")).show()
df.select("*").show()

# col expression
from pyspark.sql.functions import *
df.select(col("order_id"),col("unit_price"), col("order_status"), col("quantit
y")).show()
```

```
df.select(col("order_id"),col("unit_price"), col("order_status"), col("quantit
y")+2).show()
df.select(col("order_id"),col("unit_price"), col("order_status"), col("quantit
y")*col("unit_price")).show()
# alias
df.select(col("order_id"),col("unit_price").alias("price"), col("order_status"),
(col("quantity")+2).alias("q") ).show()

# other ways
df.select(col("order_id"),"unit_price", df["order_status"], df.quantity).show()

# selectExpr
df.selectExpr("order_id","unit_price*2 as double_q", "order_status","quantit
y+2 as q").show()
```

distinct

Used to give uniques values of a DataFrame . It is a **wide transformation.**

```
df.distinct()
df.select("state").distinct()
```

dropDuplicates

Used to **remove duplicate rows** from a DataFrame.

```
data = [
    (1, "PLACED", 100),
    (2, "SHIPPED", 200),
    (3, "DELIVERED", 300),
    (1, "PLACED", 150),
    (1, "DELIVERED", 150)
]

order_schema = "order_id int, status string, amount int"
df = spark.createDataFrame(data, order_schema)
```

```
df.dropDuplicates(["order_id","status"]).show()
```

sort / orderBy

Used to **order rows** of a DataFrame based on one or more columns.
causes a **global sort**

- **Wide transformation**
- Full shuffle
- Expensive on large datasets

```
df.orderBy("order_date")
df.sort("order_date")

# ascending descending
df.orderBy(col("order_date").desc())
df.orderBy(col("order_date").asc())

#multiple columns
df.orderBy(col("state").asc(),col("order_date").desc())

# using string method
df.sort(desc("amount")).show()
```

filter/where

Used to **filter rows** based on a condition.

They are narrow **transformations** (lazy).

```
# simple filters

df.filter(col("order_status")=="PLACED" ).show()

df.filter("order_status='PLACED'").show()
```

```
# multiple and conditions
df.filter((col("order_status")=="PLACED") &  (col("state")=="Punjab") ).sho
w()

df.filter("order_status='PLACED' and state='Punjab' ").show()

# multiple or conditions
df.filter((col("order_status")=="PLACED") |  (col("state")=="Punjab") ).sho
w()

df.filter("order_status='PLACED' or state='Punjab' ").show()

## check nulls
df.filter(col("unit_price").isNull())
df.filter(col("unit_price").isNotNull())
```

literal values

```
df.select("*",lit(1).alias("dummy")).show()

df.select("*", expr("1 as dummy")).show()
```

withColumn :

Used to **add a new column** or **modify an existing column** in a DataFrame.

They are narrow **transformations** (lazy).

```
# add a new column
df2 = df.withColumn("total sales", col("unit_price") * col("quantity")).show
()

df.withColumn("total sales", col("unit_price") * col("quantity")).withColumn
("dummy",lit(1)).show()
```

```
df.withColumn("total sales", expr("unit_price*quantity")).show()

# modify existing column
df.withColumn("quantity", lit(2) * col("quantity")).show()
```

withColumnRenamed

Used to **rename an existing column** in a DataFrame.

```
df.withColumnRenamed("order_status","status").show()
```

cast

Used to **change the data type** of a column.

```
df.withColumn("unit_price", col("unit_price").cast("int")).show()

df.select("order_id", expr("cast(unit_price as int) as unit_price")).show()

df.select("order_id", col("unit_price").cast("int")).show()
```

working with dates :

```
# default date format : YYYY-MM-DD

from pyspark.sql.functions import *
# option 1 pass date format during file read
orders_schema = 'order_id int,customer_id int,product_id int,unit_price floa
t,order_date date,order_status string,state string,quantity int'
df = spark.read.format("csv").schema(orders_schema).option("header" , "t
rue").option("dateFormat" , "yyyy/MM/dd").load("/data/orders_date_40mb.
csv")
```

```
#option 2 read it as string and later convert it to date
orders_schema = 'order_id int,customer_id int,product_id int,unit_price floa
t,order_date string,order_status string,state string,quantity int'
df = spark.read.format("csv").schema(orders_schema).option("header" , "t
rue").option("dateFormat" , "yyyy/MM/dd").load("/data/orders_date_40mb.
csv")
df.withColumn("order_date", to_date("order_date", "yyyy/MM/dd")).show()
```

when / otherwise

Used to apply **conditional logic** on columns

Equivalent to **CASE WHEN** in SQL

Basic syntax

```
from pyspark.sql.functionsimport when, col

df.withColumn(
"order_flag",
   when(col("amount") >500,"HIGH").otherwise("LOW")
)
```

Multiple conditions

```
df.withColumn(
"order_category",
   when(col("amount") >=500,"HIGH")
   .when(col("amount") >=200,"MEDIUM")
   .otherwise("LOW")
)
```

Demo

```
df.withColumn("quantity_group", when(col("quantity") == 1 , "Single Quantit
y")).show()
df.withColumn("quantity_group", when(col("quantity") == 1 , "Single Quantit
y").otherwise("Multiple quantity")).show()
df.withColumn("quantity_group", when(col("quantity") == 1 , "Single").when
(col("quantity")<=5 ,"Less than 5").otherwise("More than 5")).show()

# alternate ways
df.select("*",  when(col("quantity") == 1 , "Single Quantity").otherwise("Mul
tiple quantity").alias("qty_group") ).show()
df.select("*",  expr("case when quantity = 1 then 'single' else 'multiple' end
as qty_group")).show()
```

string and date functions

```
# string functions
# from pyspark.sql.functions import col, upper, lower, initcap, length, trim,
substring, regexp_replace, concat, lit, split
from pyspark.sql.functions import *
data = [
    (1, "ankit bansal  ", "ankit.bansal@gmail.com"),
    (2, "  rohit sharma ", "rohit@gmail.com"),
    (3, "virat kohli", "virat@bcci.in")
]

schema = ["id", "name", "email"]

df = spark.createDataFrame(data, schema)

df_string = df.select(
    "id",
    "name",
    trim(col("name")).alias("trim_name"),
    upper(col("name")).alias("upper_name"),
```

```
    lower(col("name")).alias("lower_name"),
    initcap(col("name")).alias("initcap_name"),
    length(col("name")).alias("name_length"),
    substring(col("name"), 1, 5).alias("sub_name"),
    concat(trim(col("name")), lit(" , "), col("email")).alias("concat_col"),
    split(col("email"), "@").getItem(1).alias("email_domain")
)

df_string.show(truncate=False)

# date functions

data = [
    (1, "2025-01-10"),
    (2, "2025-02-15"),
    (3, "2025-03-20")
]

schema = ["id", "order_date"]

df = spark.createDataFrame(data, schema)

df_date = df.withColumn(
    "order_date", to_date(col("order_date"), "yyyy-MM-dd")
).select(
    "id",
    "order_date",
    current_date().alias("today"),
    date_add(col("order_date"), 7).alias("plus_7_days"),
    date_sub(col("order_date"), 7).alias("minus_7_days"),
    datediff(current_date(), col("order_date")).alias("days_diff"),
    year(col("order_date")).alias("year"),
    month(col("order_date")).alias("month"),
    dayofmonth(col("order_date")).alias("day"),
    date_format(col("order_date"), "yyyy/MM/dd").alias("formatted_date")
)

df_date.show(truncate=False)
```

drop

Used to **remove columns** from a DataFrame.

```
df.drop("order_id",col("state")).show()
```

union , unionAll and unionByName

Used to **append rows** of one DataFrame to another.

Rule:

> Schemas must match exactly

- Same number of columns
- Same data types
- Same column order

Column names are **not used for matching**, only **position**.

In dataframe api : union == unionAll
Both:
Keep duplicates
Do NOT deduplicate
Generate identical execution plans
unionAll() still exists only for backward compatibility.

```
data1 = [
    (1, "PLACED", 100),
    (2, "SHIPPED", 200),
    (3, "DELIVERED", 300)
]

data2 = [
    (1, "DELIVERED", 300),
```

```
    (4, "PLACED", 150)
]

order_schema = "order_id int, order_status string, amount int"

df1 = spark.createDataFrame(data1, order_schema)
df2 = spark.createDataFrame(data2, order_schema)

df1.union(df2)

#Note : try with diffrent number of columns

# if column order is not same
data1 = [
    (1, "PLACED", 100),
    (2, "SHIPPED", 200),
    (3, "DELIVERED", 300)
]

data2 = [
    (1, 300, "DELIVERED"),
    (4, 150, "PLACED")
]

order_schema1 = "order_id int, order_status string, amount int"
order_schema2 = "order_id int, amount int, order_status string"

df1 = spark.createDataFrame(data1, order_schema1)
df2 = spark.createDataFrame(data2, order_schema2)


# option 1
df2_fixed = df2.select("order_id", "order_status", "amount")

df1.union(df2_fixed).show()

#option 2 use union by name
```

```
df1.unionByName(df2).show()

# Note : id column names are not same then it wont work

# in spark sql union will remove the duplicates
```

intersect

Returns **common rows** between two DataFrames.

```
data1 = [
    (1, "PLACED", 100),
    (2, "SHIPPED", 200),
    (3, "DELIVERED", 300)
]

data2 = [
    (3, "DELIVERED", 300),
    (4, "PLACED", 150)
]

order_schema = "order_id int, order_status string, amount int"

df1 = spark.createDataFrame(data1, order_schema)
df2 = spark.createDataFrame(data2, order_schema)
df1.intersect(df2).show()
```

exceptAll / subtract

Returns rows present in **df1 but not in df2**.

```
data1 = [
    (1, "PLACED", 100),
    (2, "SHIPPED", 200),
    (3, "DELIVERED", 300)
]
```

```
data2 = [
    (3, "DELIVERED", 300),
    (4, "PLACED", 150)
]

order_schema = "order_id int, order_status string, amount int"

df1 = spark.createDataFrame(data1, order_schema)
df2 = spark.createDataFrame(data2, order_schema)
df1.subtract(df2).show() # removes duplicates
df1.exceptAll(df2).show() # keeps duplicates
```

Simple Aggregation

Aggregation is applied on the **entire DataFrame**, producing **one row**. Wide transformation.

Examples:

- total rows → count()
- total sum → sum()
- min or max value → min() , max()

```
data_list = [
(1,'2025-01-01',100 , 'Fashion')
,(2,'2025-01-02',200 , 'Electronics')
,(3,'2025-01-01',300 , 'Electronics')
,(4,'2025-01-03',400 , 'Fashion')
,(5,'2025-01-03',400 , None)
]

table_schema = ["order_id" ,"order_date" ,"sales" , "category"]
df = spark.createDataFrame(data_list, schema = table_schema)
df.show()

df.agg(count("*").alias("totalrows"),sum("sales"),count("category") ,count
(lit(1)) ).show()
```

```
df.select(count("*").alias("totalrows"),sum("sales"),count("category") ,cou
nt(lit(1)) ).show()
```

count is a action or transformation ?

The exact distinction

1. `df.count()`

Type: DataFrame **method**

```
df.count()
```

- Belongs to `pyspark.sql.DataFrame`
- Returns a **Python integer**
- Triggers execution immediately
- This is an **ACTION**

---

2. `count()` from `pyspark.sql.functions`

Type: SQL **aggregation function**

```
from pyspark.sql.functions import count

df.select(count("*"))
```

- Returns a **Column expression**
- Used inside `select` / `agg`
- Builds a logical plan
- Needs `.show()` / `.collect()` to execute
- This is a **TRANSFORMATION**

Aggregation with group by

Used to **aggregate data per key**.

```
df.groupBy("state").agg(sum("amount"))
```

- Rows are grouped by key
- One output row per group
- Equivalent to SQL `GROUP BY`

```
data_list = [
(1,'2025-01-01',100 , 'Fashion')
,(2,'2025-01-02',200 , 'Electronics')
,(3,'2025-01-01',300 , 'Electronics')
,(4,'2025-01-03',400 , 'Fashion')
,(5,'2025-01-03',500 , 'Fashion')
]

table_schema = ["order_id" ,"order_date" ,"sales" , "category"]
df = spark.createDataFrame(data_list, schema = table_schema)

df.groupBy("category").agg(count("*").alias("totalrows"),sum("sales").alias
("total sales")).show()
df.groupBy("category","order_date").agg(count("*").alias("totalrows")\
,sum("sales").alias("total sales")).show()

# filter on aggregated data (similar to having clause)
df.groupBy("order_date").agg(count("*").alias("totalrows"),sum("sales").ali
as("total_sales")) \
.filter(col("total_sales")>300).show()
```

Joins

Used to **combine rows from two DataFrames** based on a condition.

```
orders.join(customers,"customer_id","inner")
```

Equivalent to SQL `JOIN`.

Types of joins

Join type	Description
inner	Matching rows only
left / left_outer	All left + matches
right / right_outer	All right + matches
full / full_outer	All rows
left_semi	Rows from left that have a match
left_anti	Rows from left with NO match
cross	Cartesian product

Join syntax (DataFrame API)

A. Join on same column name

```
# when join column name are same in both table
df1.join(df2,"order_id","inner")
```

Spark removes duplicate join column automatically.

B. Join with explicit condition

```
df1.join(
    df2,
    df1.order_id == df2.order_id,
"left"
)
```

Both columns retained.

Join with multiple keys

```
# when join column names are same in both tables
```

```
df1.join(
    df2,
    ["order_id","order_date"],
"inner"
)

# with explicit condition
```

Special joins

LEFT SEMI JOIN

- Returns only columns from **left**
- Used for existence check

```
orders.join(customers,"customer_id","left_semi")
```

LEFT ANTI JOIN

- Returns rows from left with **no match**

```
orders.join(customers,"customer_id","left_anti")
```

CROSS JOIN

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

- No join condition
- Very expensive
- Use with caution

Self Join (logical)

```
employee_df.alias("e").join(employee_df.alias("m")\
,col("e.manager_id")==col("m.emp_id") ,"inner" ).show()
```

demo data setup

```
from pyspark.sql.types import StructType, StructField, IntegerType, StringT
ype

# ========================
# EMPLOYEE DATAFRAME
# ========================

employee_schema = StructType([
    StructField("emp_id", IntegerType(), True),
    StructField("emp_name", StringType(), True),
    StructField("dept_id", IntegerType(), True),
    StructField("salary", IntegerType(), True),
    StructField("manager_id", IntegerType(), True),
    StructField("emp_age", IntegerType(), True)
])

employee_data = [
    (1, "Ankit", 100, 10000, 4, 39),
    (2, "Mohit", 100, 15000, 5, 48),
    (3, "Vikas", 100, 10000, 4, 37),
    (4, "Rohit", 100, 5000, 2, 16),
    (5, "Mudit", 200, 12000, 6, 55),
    (6, "Agam", 200, 12000, 2, 14),
    (7, "Sanjay", 200, 9000, 2, 13),
    (8, "Ashish", 200, 5000, 2, 12),
    (9, "Mukesh", 300, 6000, 6, 51),
    (10, "Rakesh", 500, 7000, 6, 50)
]

employee_df = spark.createDataFrame(employee_data, employee_schema)
```

```
# ========================
# DEPARTMENT DATAFRAME
# ========================

dept_schema = StructType([
    StructField("dept_id", IntegerType(), True),
    StructField("dept_name", StringType(), True)
])

dept_data = [
    (100, "Analytics"),
    (200, "IT"),
    (300, "HR"),
    (400, "Text Analytics")
]

dept_df = spark.createDataFrame(dept_data, dept_schema)

# ========================
# SHOW DATA
# ========================

employee_df.show()
dept_df.show()

# ========================
# TEMP VIEWS (OPTIONAL)
# ========================

employee_df.createOrReplaceTempView("employee")
dept_df.createOrReplaceTempView("dept")

employee_df.join(dept_df\
,employee_df.dept_id==dept_df.dept_id ,"outer" ).sort("emp_id").show()x`
```

7. NULL behavior in joins

- NULL never equals NULL

- Rows with NULL join key **do not match**
- They appear only in:
  - left join (left side)
  - right join (right side)
  - full outer join

Window Functions

1. What are window functions

Window functions perform **calculations across a set of related rows without collapsing rows**.

> groupBy → reduces rows
>
> window → keeps rows

---

2. When to use window functions

Use window functions when you need:

- Running totals
- Rankings
- Percentages of total
- De duplication with rules
- Comparisons with previous or next rows
- Aggregates per group but **row level output**

---

3. Core components of a window

A window has **three optional parts**:

```
Window.partitionBy(...)
    .orderBy(...)
    .rowsBetween(...)
```

Component	Purpose
partitionBy	Defines the group
orderBy	Defines row order inside group
frame	Defines which rows participate

---

4. partitionBy

Defines **logical groups** (similar to groupBy key).

```
Window.partitionBy("dept_id")
```

- Rows with same key form one partition
- No shuffle avoided (still distributed)
- Window functions run **within each partition**

---

5. orderBy

Defines **ordering inside each partition**.

```
Window.partitionBy("dept_id").orderBy("salary")
```

Required for:

- Ranking functions
- Running aggregates
- Lead and lag

---

6. rowsBetween

Definition

Frame based on **row position**, not values.

```
Window.rowsBetween(Window.unboundedPreceding, Window.currentRow)
```

Meaning:

> From first row in partition
>
> up to current row index

Use cases

- Running sum
- Cumulative count
- Deterministic calculations

---

7. Default window behavior

If you write:

```
Window.partitionBy("dept").orderBy("salary")
```

Spark assumes:

```
ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
```

This often surprises people.

---

8. Common window functions

Aggregation functions

```
sum, avg, min, max, count
```

Example:

```
sum("salary").over(w)
```

Ranking functions

```
row_number()
rank()
dense_rank()
```

Function	Behavior
row_number	Unique sequential number
rank	Gaps in ranking
dense_rank	No gaps

Analytical functions

```
lead()
lag()
first()
last()
```

Example:

```
lag("salary", 1).over(w)
```

9. Common patterns

Running total

```
w = Window.partitionBy("dept_id") \
        .orderBy("salary") \
        .rowsBetween(Window.unboundedPreceding, Window.currentRow)

df.withColumn("running_salary", sum("salary").over(w))
```

Percentage of total (global)

```
w = Window.partitionBy()
```

```
df.withColumn("total_salary", sum("salary").over(w)) \
  .withColumn("perc", col("salary") * 100 / col("total_salary"))
```

Deduplication (best practice)

```
w = Window.partitionBy("emp_id").orderBy(col("update_ts").desc())

df.withColumn("rn", row_number().over(w)) \
  .filter(col("rn") == 1) \
  .drop("rn")
```

Top N per group

```
w = Window.partitionBy("dept_id").orderBy(col("salary").desc())

df.withColumn("rn", row_number().over(w)) \
  .filter(col("rn") <= 3)
```

---

10. Performance characteristics (CRITICAL)

- Window functions are **wide transformations**
- Cause shuffle on partitionBy
- Require sorting on orderBy
- Memory intensive
- Can spill to disk

Window vs groupBy

groupBy	window
Reduces rows	Preserves rows
One row per group	Same rows
Simple aggregation	Analytical

groupBy	window
Cheaper	More expensive

SQL equivalent

```
SUM(salary) OVER (
  PARTITION BY dept_id
  ORDER BY salary
  ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
)
```

Same execution plan.

UDF (User Defined Functions):

1. What is a UDF

UDF (User Defined Function) allows you to define your own custom function and use it in Spark just like a built in function.

2. How to create a Python UDF

Step 1: Define a normal Python function

```
def order_priority(order_status, state, unit_price, quantity):
    order_value = unit_price * quantity

    if order_status == "CANCELLED":
        return "IGNORE"

    if state in ("Delhi", "Haryana", "Punjab"):
        if order_value >= 5000:
            return "P1"
        else:
            return "P2"
```

```
    else:
        if order_value >= 10000:
            return "P1"
        else:
            return "P3"
```

Step 2: Convert it into a Spark UDF

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

order_priority_udf = udf(order_priority, StringType())
```

Here:

- `order_priority` → Python function
- `StringType()` → return type of UDF

3. How to use UDF in DataFrame API

```
from pyspark.sql.functions import col

df.withColumn(
    "order_priority",
    order_priority_udf(
        col("order_status"),
        col("state"),
        col("unit_price"),
        col("quantity")
    )
).show()
```

UDF behaves like a normal Spark function.

4. How to register UDF for Spark SQL

Step 1: Register the UDF

```
spark.udf.register("order_priority_sql_udf", order_priority, StringType())
```

Step 2: Use it in Spark SQL

```
df.createOrReplaceTempView("orders")

spark.sql("""
    SELECT
        order_id,
        state,
        unit_price,
        quantity,
        order_status,
        order_priority_sql_udf(order_status, state, unit_price, quantity) AS orde
r_priority
    FROM orders
""").show()
```

complete code

```
from pyspark.sql.functions import *
from pyspark.sql.types import *
orders_schema = 'order_id int,customer_id int,product_id int,unit_price floa
t,order_date date,order_status string,state string,quantity int'
df = spark.read.format("csv").schema(orders_schema).option("header" , "t
rue").load("/data/orders_1gb.csv")

def order_priority(order_status, state, unit_price, quantity):
    order_value = unit_price * quantity

    if order_status == "CANCELLED":
        return "IGNORE"

    if state in ("Delhi", "Haryana", "Punjab"):
        if order_value >= 300:
```

```
            return "P1"
        else:
            return "P2"
    else:
        if order_value >= 500:
            return "P1"
        else:
            return "P3"


order_priority_udf = udf(order_priority, StringType())

df.withColumn(
    "order_priority",
    order_priority_udf(
        col("order_status"),
        col("state"),
        col("unit_price"),
        col("quantity")
    )
).show(100)

spark.udf.register("order_priority_sql_udf", order_priority, StringType())

df.createOrReplaceTempView("orders")

spark.sql("""
    SELECT
        order_id,
        state,
        unit_price,
        quantity,
        order_status,
        order_priority_sql_udf(order_status, state, unit_price, quantity) AS orde
r_priority
    FROM orders
""").show()
```

transformations and actions list

tranformation / actions
T → Transformations
→Transform data
→Tell Spark what to do
→Task is planned
→Time is NOT spent yet (lazy)

list of transformations
select
selectExpr
filter
where
withColumn
withColumnRenamed
drop
dropDuplicates
distinct
groupBy
agg
join
union
intersect
subtract
orderBy
sort
limit
repartition
coalesce

A → Actions
→Actually run
→Activate job

→Ask Spark for a result
→Action = execution

list of actions
show
collect
count
take
first
head
tail
save
saveAsTable
insertInto
write (parquet, csv, json, orc, format, etc)