sort() / orderBy()

- we can use either sort() or orderBy() function of pyspark Dataframe by ascending / desiending order based on single or multiple columns.

⇒ df.sort("col1") ; df.sort("col1", "col2")

⇒ df.orderBy("col1") ; df.orderBy("col1","col2")

df.sort(df.salary.desc())

df.sort(col("col1").desc())

df.orderBy(col("col").desc())

df.orderBy(col("col").asc())

GroupBy

- It allows you to perform aggregate functions on groups of rows, rather than on individual rows, enabling you to summarize data & generate aggregate statistics.

aggregate functions (agg())

count() mean() max() min()

sum() avg()

similar to groupBy in SQL

⇒ df.groupBy("col1")

```
df.groupBy("col1").count()
```

```
df.groupBy("col1") \
  .agg(
      sum("col2").alias("____"),
      avg("col2").alias("____"),
      max("col3").alias("____")
  )
```

```
df.groupBy("col1") \
  .agg(
      sum("col2").alias("____")
  ) \
  .filter("____") \
  .orderBy("____") \
  .limit(5)
```

groupBy + having

```
df.groupBy("col1") \
  .agg(
      sum("col2").alias("____")
  ) \
  .filter("____")
```

Joins

- Pyspark joins is used to combine two dataframes
- Pyspark join operation combies data from two or more datasets based on a common column or key.

Inner Join,	Full outer join	cross join
left join,	left semi join	self join
right join,	left anti join	

All join types in PySpark

Standard joins	Special joins	Advanced joins
① Inner join	⑤ left semi join	⑦ Cross join (Cartesian join)
② Left join (left outer join)	⑥ left anti join	⑧ self join
③ right join (right outer join)	NO; right semi ✗ right anti ✗	⑨ BroadCast join (optimization technique not a join type)
④ Full outer join		

① Inner Join :

- It returns only the rows that have matching keys in Both tables

→ Match exists on both side : keep row
No match on either side : drop row

orders

o_id	cust_id	amt
1	101	500
2	102	700
3	103	300

customers

cust_id	country
101	India
103	USA
104	UK

Inner join Syntax

① same column name

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

② Different column name

```
df_orders.join(
    df_customers,
    df_orders.cust_id == df_customers.id,
    "inner"
)
```

o/p

o_id	cust_id	amt	country
1	101	500	India
3	103	300	USA

cust_id = 102 ✗ Normal ch
cust_id = 104 ✗

② Left join (left outer join)

- returns ALL rows from the left table, & only matching rows from the right table
- If there's no match on the right → NULLs are filled

```
orders.join(customers, "cust_id", "left")
```

```
orders.join(
        customers,
        orders.cust_id == customers.cid,
        "left"
)
```

o/p

o_id	cust_id	amt	country
1	101	500	India
2	102	700	NULL
3	103	300	USA

③ right join (right outer join)

- returns ALL rows from the right table, & only matching rows from the left table
- If there's no match on left → NULLs are filled.

```
orders.join(customers, "cust_id", "right")
```

```
orders.join(
        customers,
        orders.cust_id == customers.cid,
        "right"
)
```

o/p:

cust_id	country	or_id	amt
101	India	1	500
103	USA	3	300
104	UK	NULL	NULL

④ Full Outer Join :

- returns ALL rows from both tables
- If a row has no match on the other side → missing columns become NULL

Orders

o_id	cust_id	amt
1	101	500
2	102	700
3	103	300

Customers

cust_id	Country
101	India
103	USA
104	UK

Syntax:

```
orders.join(customers, "cust_id", "outer")
```

```
orders.join(
        customers,
        orders.cust_id == customers.id,
        "outer"
    )
```

o/p:

o_id	cust_id	amt	Country
1	101	500	India
2	102	700	NULL
3	103	300	USA
NULL	104	NULL	UK

⑤ Left semi:

- Left Semi Join retorns rows from Left table that having a match in the right table
- NO columns from the right table are retorned.

Syntax:

```
ordersjoin(customers, "cust_id", "left_semi")
```

```
orders.join(
        customers,
        orders.cust_id == customers.id,
        "left_semi"
    )
```

left-semi- o/p:

o_id	cust_id	amt
1	101	500
3	103	300

{returns only common rows that are from on only left-table}

inner join - o/p:

o_id	cust_id	amt	country
1	101	500	India
3	103	300	USA

{returns common rows & respective all columns from both tables}

⇒ similar to Exists in SQL

```
select * from orders o
where exists (
        select 1
        from customers c
        where o.cust_id = c.cust_id
    )
```

⑥ Left Anti Join

- returns rows from the left table that do not have a match in right table
- No columns from the right table are returned

orders

o_id	cust_id	amt
1	101	500
2	102	700
3	103	300

customers

cust_id	country
101	India
102	USA

Syntax:

```
orders.join(customers, 'cust_id', 'left-anti')
```

```
orders.join(
        customers,
        orders.cust_id == customers.id,
        "left-anti"
)
```

o/p:

o_id	cust_id	amt
2	102	700

left semi → rows that have a match
left anti → rows that do not have a match

In SQL;

semi = exists
anti = not exists

SQL similar Code;

```
select *
from orders o
where not exists (
        select 1
        from customers c
        where o.customer_id = c.customer_id
)
```

---

⑦ cross Join: (Cartesian Join)

returns every possible combination of rows from both tables

(left rows × right rows = output rows)

```
orders.crossJoin(customers)
```

---

⑧ self Join:

- is when a dataframe is joined with itself to compare rows within the same table

Note:

same table, different roles

You treat one Dataframe as two logical tables using aliases

employees

emp_id	emp_name	manager_id
1	A	NULL
2	B	1
3	C	1
4	D	2

Syntax:

```
e = employees.alias("e")
m = employees.alias("m")

result = e.join(
            m,
            e.manager_id = m.emp_id,
            "left"
        )
        .select(
            e.emp_name.alias("employee"),
            m.emp_name.alias("manager")
        )
```

Union:

→ union is used to append rows of one dataframe to another

[vertical combine, not side-by-side like joins]

{ union → stack rows
  join → add columns }

① Union:

- combines rows from both dataframes
- keeps duplicates
- (column order & data types must match)

syntax: `df1.union(df2)`

df1

id	name
1	A
2	B

df2

id	name
2	B
3	C

o/p

id	name
1	A
2	B
2	B
3	C

- Duplicates are kept

What if?

df1

id	name
1	A

df2

name	id
B	2

df1.union(df2)

id	name
1	A
B	2

df1 : id, name ; df2 : name, id

df1.union(df2)

⇒ Data corruption happens silently.
⇒ NO error; but values are swapped

Why spark does Not throw error?

① Column count ⇒ same
② Data types compatible
③ spark does not check columns names in unions

⇒ how data types compatible?

df1 : id : int; name : string ⇒ (1, 'A')

df2 : name : string, id : int ⇒ ('B', 2)

df1.union(df2)

key Rule :
spark finds a common super type for each column position

⇒ spark compares column by position, not by name

df1

position 0 → id : int
position 1 → name : string

df2

position 0 → name : string
position 1 → id : int

spark compares columns by position; not by name

position 0 ⟹ df1 → int
df2 → string
} common super type → "string"

spark casts int → string

position 1 ⇒ df1 → name } string
df2 → id } int
common super type "string"

Therefore;

df1.union(df2)

position 0 → string
position 1 → string.

we can check this by .printSchema()

⇒ Due to this problem with union
we basically prefer the (unionByName())

union() →
- compares column by position not by names
- kept duplicates

problems with union()

unionAll() ⟶ (deprecated)

- `unionAll()`
  does not exist as a separate behavior in modern pyspark.

***
In pyspark DataFrame API,
union() → already behaves like SQL UNION ALL
unionAll() → is deprecated/alias and should not be used

In SQL;
union → removes duplicates after combining rows
union All → keeps duplicates

In pyspark; present version:
- union() → keeps duplicates
- ~~unionAll()~~ → deprecated

⟶

If we want unique rows after union()

```
df1.union(df2).distinct()
```

↳ removes duplicate rows

unionByName()

- safe & correct way to union dataframes

⇒ unions dataframes by matching column names instead of column positions

⇒ This avoids silent data corruption.

df1

id	name
1	A

df2

name	id
B	2

df1.unionByName(df2)

id	name
1	A
2	B

⇒ allowMissingColumns = True

if; df1: (id, name, salary)
df2: (id, name)

→ without allowMissingColumns

```
df1.unionByName(df2) # Error.
```

⇒ with allowMissingColumns

```
df1.unionByName(df2, allowMissingColumns = True)
```

id	name	salary
1	A	50000
2	B	NULL

⊗ Column name mismatch = error
{id vs emp-id}

Note:

→ Column names must match
- case-sensitive
- spaces matter

→ Data types must be compatible
- Spark will promote types if needed