Rank window functions

- row_number()
- rank()
- dense_rank()
- percent_rank()
- ntile(n)
- cume_dist()

row_number():

- assigns a unique, sequential numbers to each row within a window
- Starts from 1
- No duplicates
- Even tied values get different numbers

→ It only cares about row position after sorting

row_number() →

must be used with
- over()
  - orderBy()

(otherwise numbering is meaningless)

Sample data:

emp	dept	salary
A	IT	1000
B	IT	1500
C	IT	1500
D	HR	900
E	HR	1100

① Basic "row_number()" (global ranking):

```
df.withColumn(
        'rn',
        row_number().over(
                Window.orderBy('salary')
        )
)
```

o/p:

emp	dept	salary	rn
D	HR	900	1
A	IT	1000	2
E	HR	1100	3
B	IT	1500	4
C	IT	1500	5

② row_number() with "partitionBy"

```
df.withColumn(
        'rn',
        row_number().over(
                Window.partitionBy('dept')
                .orderBy('salary')
        )
)
```

o/p:

emp	dept	Salary	rn
A	IT	1000	1
B	IT	1500	2
C	IT	1500	3
D	HR	900	1
E	HR	1100	2

If values are equal row_number() still increments

> order among tees is not guaranteed

③ Controlling tie orders.

```
df.withColumn(
        'rn',
        row_number().over(
            Window.partitionBy('dept')
                .orderBy('salary', 'emp')
        )
)
```

④ Top-N per group:

i.e. (Top 2 salaries per department)

```
top= df.withColumn(
        'rn',
        row_number().over(
            Window.partitionBy('dept')
                ~~.orderBy('salary').desc()~~
                .orderBy(col('salary').desc())
        )
)
top.filter(col('rn') <= 2).show()
```

Why row_number() here?

- guarantees exactly N rows
- No ambiguity with ties

"row_number()"

↓

Best for : {Dedup, top-N}

Deduplication:

Window.partitionBy('id').orderBy(col('updated_at').desc())

id	name	updated_at
101	A	2024-01-01
101	A	2024-01-05
101	A	2024-01-10
102	B	2024-01-03
102	B	2024-01-08

This is how the duplicates are droped using "row_number()"

```
df = df.withColumn(
        'rn',
        row_number().over(
            Window.partitionBy('id')
            .orderBy(col('updated_at').desc())
        )
)
```

```
df.filter(col('rn') == 1)
```

id	updated_at	rn
101	2024-01-10	1
101	2024-01-05	2
101	2024-01-01	3
102	2024-01-08	1
102	2024-01-03	2

id	updated_at	rn
101	2024-01-10	1
102	2024-01-08	1

rank():

- assigns a rank to rows within a window based on ordering, & it handles ties by giving the same rank

- Ranks can skip numbers
  {tied values → same rank}

Sample data

emp	dept	salary
A	IT	1000
B	IT	1500
C	IT	1500
D	HR	900
E	HR	1100

{ . same rank
. gaps available
↓
i.e. gap-ranking }

Ex:
Olympic medal assigning;
if [redacted] & people score high & same they are given with Gold; 3rd will be Bronze

1) Basic rank():

```
df.withColumn(
    'rn',
    rank().over(
        Window.orderBy('salary')
    )
)
```

emp	dept	salary	rn
D	HR	900	1
A	IT	1000	2
E	HR	1100	3
B	IT	1500	4
C	IT	1500	4

2) rank() with partitionBy:

```
df.withColumn(
    'rn',
    rank().over(
        Window.partitionBy('dept')
              .orderBy('salary')
    )
)
```

Emp	dept	salary	rn
A	IT	1000	1
B	IT	1500	2
C	IT	1500	2
D	HR	900	1
E	HR	1100	2

3. Top-N with rank()

Top 2 salaries per department

```
df = df.withColumn(
    'rn',
    rank().over(
        Window.partitionBy('dept')
        .orderBy(col('salary').desc())
    )
)
df.filter(col('rn') <= 2)
```

---

rank() vs row_number()

features	row_number()	rank()
unique number	Yes	No
Tie handling	X	✓
Gaps in ranking	X	✓
Best use	Dedup, exact N	LeaderBoards

Real-world production use case of "rank()"

① **Leaderboards:**
- Top performances
- Top selling product
- Exam ranks

② **Reporting:**
- Show all employees with 2nd highest salary
- Top 3 scores including ties

Ex:- Olympics medal ranking is a classic example for "rank()".

dense_rank():

- assigns ranks to rows within a window, gives the same rank to ties, & does not skip ranks

·NO gaps in ranking·

① Basic "dense_rank()"

```
df.withColumn(
        "dr",
        dense_rank().over(
                Window.partitionBy
                        .orderBy('salary')
        )
)
```

o/p

emp	dept	salary	dr
D	HR	900	1
A	IT	1000	2
E	HR	1100	3
B	IT	1500	4
C	IT	1500	4

: 5

② dense_rank() with partitionBy:

```
df.withColumn(
        "dr",
        dense_rank().over(
                Window.partitionBy('dept')
                        .orderBy('salary')
        )
)
```

emp	dept	salary	dr
A	IT	1000	1
B	IT	1500	2
C	IT	1500	2
D	HR	900	1
E	HR	1100	2

rank() vs dense_rank()

salary	rank	dense_rank
1000	1	1
1500	2	2
1500	2	2
1600	4	3

rank() → skips

dense_rank() → continues

Ex:

1) School Exam ranking:

- Two students tie for 1st
- Next student is 2nd, not 3rd

This is (dense_rank())

Real-world production use cases

1) Reporting dashboards
   - Top categories
   - Top-selling products
   - Salary bands

2) Analytics
   - Distinct ordering
   - Bucketed ranking

Percent_rank():

tells you the relative position of a row as a percentage within a window

output range : 0.0 → 1.0
First row : 0.0
Last row : 1.0

Formula: (Interview Imp)

percent_rank = (rank - 1) / (total_rows - 1)

- Uses rank() Internally
- That's why ties behave like rank() {with gaps}

① Basic "percent_rank()"

```
df.withColumn(
        'pr',
        percent_rank().over(
                Window.orderBy('salary')
        )
)
```

emp	salary	rank	percent_rank
D	900	1	0.0
A	1000	2	0.25
E	1500	3	0.50
B	1500	4	0.75
C	1500	4	0.75

② percent_rank() with partitionBy

```
df.withColumn(
        'pr',
        percent_rank().over(
            Window.partitionBy('dept')
                  .orderBy('salary')
        )
)
```

Emp	Salary	dept	PR
A	1000	IT	0.00
B	1500	IT	0.50
C	1500	IT	0.50
D	900	HR	0.00
E	1100	HR	1.00

percent_rank() (vs) cume_dist()

Feature	percent_rank	cume_dist
Formula-based	✓	✗
First row	0.0	> 0
Last row	1.0	1.0
Tie handling	rank-based	count-based
Best for	Relative position	coverage

Real-world use cases:

1. Distributed analysis
   - salary percentiles
   - score normalization
2. Threshold filtering

   {percent_rank <= 0.1} → (top 10%)

ntile(n):

- divides the rows in a window into 'n' approximately equal buckets & assigns a bucket number to each row.
  - Buckets are numbered 1 to n
  - rows are distributed as evenly as possible.
  - order matters.

- split ordered data into N groups.

data

emp	dept	salary
A	IT	1000
B	IT	1200
C	IT	1500
D	IT	1800
E	IT	2000

(total rows = 5)

① Basic ntile(2)

```
df.withColumn(
    "bucket",
    ntile(2).over(
        Window.orderBy('salary')
    )
)
```

emp	salary	bucket
A	1000	1
B	1200	1
C	1500	1
D	1800	2
E	2000	2

How spark distributes rows?

Rules:

1. Rows are ordered
2. rows are divided as evenly as possible.
3. Earlier buckets get extra rows.

Note:

First bucket may have more rows if rows aren't divisible evenly

② ntile(4):

```
df.withColumn(
    'bucket',
    ntile(4).over(
        Window.orderBy('salary')
    )
)
```

o/p:

emp	salary	bucket
A	1000	1
B	1200	1
C	1500	2
D	1800	3
E	2000	4

③ ntile(n) with partitionBy:

```
df.withColumn(
    'bucket',
    ntile(2).over(
        Window.partitionBy('dept')
              .orderBy('salary')
    )
)
```

o/p:

emp	dept	salary	bucket
A	IT	1000	1
B	IT	1200	1
C	IT	1200	2
D	HR	900	1
E	HR	1100	2

⇒ Buckets are calculated independently per department

key behaviours:

- Uses row positions (not values)
  - salary gaps don't matter
- orderBy is mandatory
  - without order → buckets meaningless
- Bucket sizes differ by at most 1
  - Balanced distribution

Real-world use cases:

① customer segmentation
- Top 25%, middle 50%, bottom 25%

② salary bands
- Quartiles/deciles

③ Data samples
- Evenly split data

Cume_dist():

→ returns the cumulative distribution of a row within a window

{What fraction of rows have values less than or equal to the current row?}

⟶ output range: (0,1]
- last row is always 1.0
- Ties are handled naturally

How cume_dist() is calculated?

$$\text{cume\_dist} = \frac{\text{number of rows <= current row}}{\text{total rows}}$$

① Basic cume_dist()

```
df.withColumn(
    'cd',
    cume_dist().over(
        Window.orderBy('salary')
    )
)
```

Sample Data:

emp	salary
D	900
A	1000
E	1100
B	1500
C	1500

Total rows = 5

emp	salary	cd
D	900	0.20
A	1000	0.40
E	1100	0.60
B	1500	1.00
C	1500	1.00

Why both B and C are 1.00?

- All 5 rows have salary <= 1500

---

cume_dist()

tells: up to this row, how much of the data is already covered?

salaries (sorted): 900, 1000, 1100, 1500, 1500

(Total rows = 5)

→ row by row thinking:

Row 1 → salary = 900

- Rows <= 900 → 1
- Total rows → 5

cume_dist = 1/5 = 0.20

Row 2 → salary = 1000

- rows <= 1000 → 2

cume_dist = 2/5 = 0.40

Row 3 → salary = 1100

- rows <= 1100 = 3

cume_dist = 3/5 = 0.60

row 5 → salary = 1500

rows <= 1500 ⇒ 5

cume_dist = 5/5 = 1.00

Row 4 → salary = 1500

- rows <= 1500 = 5

Why 5?

Because two rows have salary = 1500

cume_dist = 5/5 = 1.00

② cume_dist() with partitionBy

```
df.withColumn(
    'cd',
    cume_dist().over(
        Window.partitionBy('dept')
        .orderBy('salary')
    )
)
```

df:

emp	dept	salary	cume_dist	
A	IT	1000	0.33	(1/3)
B	IT	1500	1.00	(3/3)
C	IT	1500	1.00	(3/3)
D	HR	900	0.50	(1/2)
E	HR	1100	1.00	(2/2)

Feature	percent_rank()	cume_dist()
Formula	(rank - 1) / (n - 1)	rows <= current / n
First row	0.0	> 0
Last row	1.0	1.0
Tie Behaviour	rank-based	count-based
Best for	relative position	coverage

Real-world use cases:

① percentile thresholds
(cume_dist <= 0.8)
→ Bottom 80%

② Distribution analysis
- sales distribution
- scores coverage